

CLEAN JAVASCRIPT

JS

A CONCISE GUIDE TO LEARNING CLEAN
CODE, SOLID AND UNIT TESTING

MIGUEL A. GÓMEZ

Clean JavaScript. English Edition

A concise guide to learning Clean Code, SOLID
and Unit Testing
Software Crafters

This book is for sale at <http://leanpub.com/cleanjavascript>

This version was published on 2021-06-09



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2021 Software Crafters

To Eliza, for her infinite patience.

Contents

Preface	1
What this book is not.	1
About the author	2
Introduction	3
Technical Debt	5
Kinds of technical debt	5
Refactoring, debts are paid	7
Prevention is better than cure, the rules of simple design	7
Section I: Clean Code	9
What is Clean Code?	10
Variables, names and scope	12
Correct use of <i>var</i> , <i>let</i> y <i>const</i>	12
Pronounceable and expressive names	14
Absence of technical information in the names	14
Establish a ubiquitous language	14
Names by data type	15
Numbers	16
Scope of the variables	18
Functions	23
Function declaration	23
Function Expression	23
Function expression with the arrow function	24

CONTENTS

Immediately-invoked Function Expressions (IIFE)	27
Parameters and arguments	28
Indentation size and levels	33
Declarative vs imperative style	36
Anonymous functions	38
Referential transparency	38
DRY Principle	39
Command-Query Separation (CQS)	41
Efficient algorithms	42
Classes	45
Prototype and modern ECMAScript	45
Reduced size	49
Organization	52
Prioritize composition over inheritance	53
Comments and format	56
Avoid using comments	56
Consistent format	56
Section II: SOLID Principles	58
Introduction to SOLID	59
From STUPID to SOLID	60
What is code smell?	60
Singleton pattern	60
Tight Coupling	61
Premature optimization	62
Indescriptive Naming	63
Duplication	64
SOLID principles to the rescue	65
Single responsibility principle (SRP)	66
What do we understand by responsibility?	66
Applying the SRP	68
Detecting violations of SRP	70

CONTENTS

OCP - Open/Closed principle	72
Applying OCP	72
Adapter pattern	73
Detect violations of OCP	76
LSP - Liskov Substitution Principle	77
Applying LSP	77
Detect violations of LSP	82
ISP - Interface segregation principle	83
Applying the ISP	83
Detect violation of ISP	88
DIP - Dependency Inversion Principle	89
High-level modules and low-level modules	89
Depending on Abstractions	91
Dependency Injection	91
Applying DIP	93
Detecting violations of DIP	95
Section III: Testing & TDD	96
Introduction to testing	97
Types of software tests	99
What do we understand by testing?	99
Manual tests vs automatic tests	99
Functional vs non-functional tests	99
Non-functional tests	100
The Pyramid of testing	101
Ice Cream cone anti-pattern	103
Unit testing	104
Characteristics of the unit tests	104
Anatomy of a unit test	105
Jest, the definitive JavaScript testing framework	108
Features	108

CONTENTS

Installation and configuration	109
Our first test	110
Asserts	112
Organisation and structure	113
State management: before and after	114
Code coverage	115
TDD - Test Driven Development	117
The three laws of TDD	117
Red-Green-Refactor cycle	117
TDD as a design tool	119
Implementation strategies, from red to green	119
TDD Limitations	124
Practical TDD: The FizzBuzz Kata	125
The code katas	125
The Fizzbuzz kata	126
Exercise statement	126
Designing the first test	126
We run and... red!	128
We go green	129
Adding new tests	130
Refactoring the solution, applying pattern matching.	134
References	137

Preface

JavaScript is now one of the most popular programming languages in the world. It is used in the critical infrastructures of many very important companies like Facebook, Netflix or Uber.

This is why writing better, higher quality and clearly legible code has become essential. Normally, developers write code without the explicit intention of making it easily understood by other people because we are focused on finding a solution that works and solves the problem without thinking about people. Usually, trying to understand the code written by other programmers or even code that we wrote

ourselves a few weeks ago, can be quite difficult.

This small e-book strives to be a concise reference of how to apply clean code, SOLID, unit testing and TDD, to be able to write more legible, maintainable and change tolerant JavaScript code. In this book you will find several references to other authors and simple examples that will help you to find the way to become a better developer.

What this book is not.

Before you buy this book, I have to tell you that its aim is not to teach programming from scratch, but I want to deal with some fundamental questions related to good practices for improving your JavaScript code and explain them clearly and concisely.

About the author

My name is Miguel A. Gómez, I was born in Tenerife (Canary Islands) and I studied Electronic Engineering and Computer Engineering. I consider myself to be a Software Craftsman who avoids the typical dogmatism of the craftsmanship community. I am also very interested in software development with Haskell.

Currently I'm working as a Senior Software Engineer in an American startup. This company develops software solutions for the legal profession, I have participated in several projects as a main developer.

Amongst the most important positions I have held are those of multi-platform mobile developer with Xamarin and C# and Full Stack developer (which is my current job). Here, I apply a hybrid programming style mixing Object Oriented Programming and Functional Reactive Programming, in the frontend side with TypeScript, RxJS and ReactJS, in the backend side with TypeScript, NodeJS, RxJS and MongoDB and I also manage DevOps process with Docker in Azure.

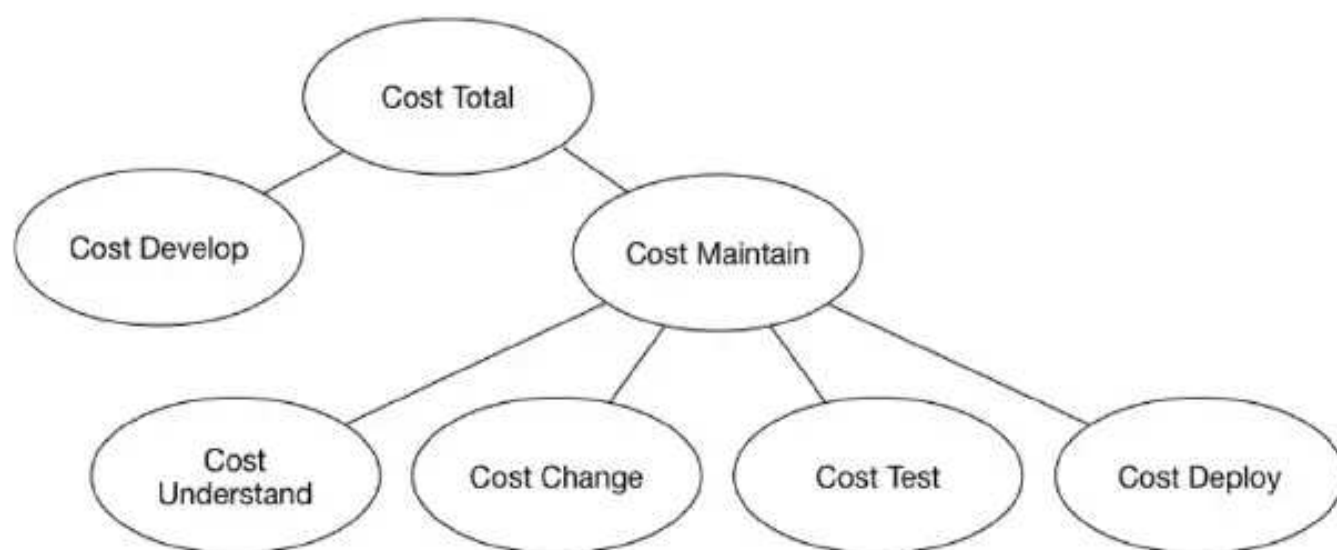
Apart from this, I am the co-founder of [Omnirooms.com](https://omnirooms.com). This is a startup that aspires to remove the barriers that disabled people find when they try to book their vacations. I am also the founder of softwarecrafters.io, which is a Software Craftsmanship, DevOps and software technology community.

Introduction

“The strength of JavaScript is that you can do anything. The weakness is that you will”. Reginald Braithwaite

Over recent years, JavaScript has become one of the most widely used languages around the world. The main advantage and at the same time its biggest weakness is its versatility. This great versatility has resulted in some bad practice that has spread through the community. Even so, you can find JavaScript in the critical infrastructures of some very important companies. Facebook, Netflix and Uber use it. Limiting the cost of software maintenance is essential in these kinds of companies.

The total cost of a software product is found by adding the development and maintenance costs. A lot of developers don't realise that the maintenance costs normally end up being higher than the initial development cost. Besides, as Kent Beck explains in his book “Implementation Patterns”, the maintenance cost is given by the sum of the cost of understanding the code, changing it, testing it and deploying it.



Cost of a software product

In addition to the aforementioned costs, Dan North, famous for being one of the creators of BDD, also includes the opportunity cost and the cost related to the delay in the releases. Although I will not comment on topics related to project

management in this book, I do think it is important to be aware of all the costs that the developers generate, and especially to try to minimize these wherever we can. In the first part of the book I will try to present some ways of minimising the costs related to understanding the code. I will try to synthesize and extend some of the concepts related to this that [Robert C. Martin](https://twitter.com/unclebobmartin)¹, [Kent Beck](https://twitter.com/KentBeck)², and [Ward Cunningham](https://twitter.com/WardCunningham)³ put forward about Clean Code and which other authors apply to JavaScript. I will also address some of my own language concepts, which once comprehended should help us design better software.

In the second part we are going to see how the SOLID principles can help in writing much more intuitive code which will help to reduce the maintenance costs related to changing the code.

In the third and the last part, we are going to talk about how unit testing and test driven development (TDD) can help us to write better and more robust code. This helps us to prevent technical debt and minimize the costs related to software testing.

¹<https://twitter.com/unclebobmartin>

²<https://twitter.com/KentBeck>

³<https://twitter.com/WardCunningham>

Technical Debt

“A Lannister always pays his debts” - Game of Thrones

We could regard technical debt as a metaphor for explaining how the lack of quality in the code of a software project generates a debt which can cause future cost overruns. These costs are directly related to the capacity of the project to accept change.

The concept of technical debt was first introduced by Ward Cunningham in the OOPSLA conference in 1992. Since then, different authors have been trying to extend the metaphor to cover more economic concepts and other situations in the life cycles of software.

Kinds of technical debt

According to Martin Fowler, the technical debt can be classified into four kinds depending on its origin, as illustrated by this technical debt quadrant:

	Reckless	Prudent
Deliberate	"We don't have time for design"	"We must ship now and deal with consequences"
Inadvertent	"What's layering?"	"Now we know how we should have done it"

Technical debt quadrant

- **Reckless and deliberate debt:** In this kind of debt the behavior of the developer is conscious and irresponsible. This normally results in a low quality project with a high maintenance difficulty.
- **Reckless and inadvertent:** This kind of debt is probably the most dangerous because it is generated from ignorance and lack of experience, usually by a junior developer but in the worst cases, by a false senior.
- **Prudent and deliberate:** Ward Cunningham said that this kind of debt could be good for accelerating project development, as long as we pay it off as soon as possible. The danger with leaving debt unpaid is that the longer we have incorrect code, the more the interest will increase.
- **Prudent and inadvertent:** In the majority of projects this kind of debt is common. Since this is related to the knowledge acquired by the programmer himself throughout the development of the project, a moment arises when he becomes conscious that he could have chosen a better design. This is the moment when it is necessary to evaluate whether the acquired debt should be paid or whether it can be postponed.

Refactoring, debts are paid

As we have seen, despite its negative repercussions, incurring technical debt is sometimes unavoidable. In this case, we should ensure that we are aware of the implications and try to pay the debt as soon as possible, like a good Lannister. But, how does a developer pay the technical debt? The short answer is: by refactoring.

Refactoring is a process that aims to improve the code of a project without changing its behaviour, in order to make it more understandable and maintainable.

To start refactoring, it is essential that our project has automatic tests, either unit or integration tests, that allow us to know in every moment if the code that we change is still meeting requirements. Without automatic tests the refactoring process is too complicated, so we will probably end up adopting defensive attitudes like “if it works, don’t fix it”, due to the implicit risk involved in modifying the system.

If the project has automatic tests, the next step is knowing when we should refactor our code. Generally, we should refactor when we notice that the code doesn’t have enough quality, or when we detect some code smell, we will see a few examples of these in the next chapters. Personally I like to do daily refactoring. I think it is a good warm-up, it is a good way to reconnect with work done the day before.

Sometimes, refactoring might not be enough and we might need to change the architecture or redesign some components. For example, if we imagine that we have a NodeJS project that should be very scalable and accept a large amount of concurrent requests. It would probably be a good idea to design a queue system for it. However the team could prudently and deliberately decide to release the first version with a simpler architecture in order to validate the product in the market even while knowing that the architecture will need to be changed in the short or medium term.

Prevention is better than cure, the rules of simple design

Bad quality design in software always ends up being paid for, and someone ends up being liable, either the client, the supplier with their resources, or the developer

who spends more time refactoring or wastes time programming over a fragile system. This is why prevention is better than cure.

A good starting point for anticipating technical debt is to try to assess whether we are respecting the four rules of simple design created by Kent Beck:

- The code successfully passes the tests
- The code reveals design intention (should be easy to understand)
- No duplication (DRY)
- Fewest possible components (remove anything that doesn't serve the three previous rules)

Throughout this book we will be looking at how to apply Clean Code, SOLID, TDD and many other associated concepts that will help us try to comply with these four rules.

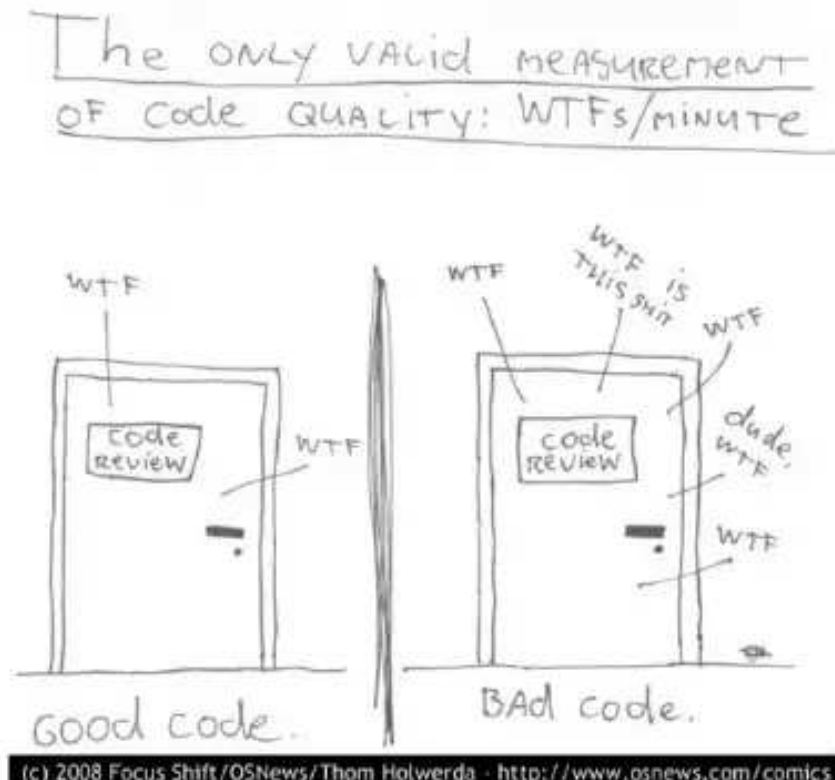
Section I: Clean Code

What is Clean Code?

“Programming is the art of telling another human being what one wants the computer to do.” – Donald Knuth

Clean Code is a term that was already being used by developers such as Ward Cunningham and Kent Beck, although it didn't catch on until Robert C. Martin, aka Uncle Bob, published his book “Clean Code: A Handbook of Agile Software Craftsmanship” in 2008.

This book, although it is a bit dogmatic and maybe too focused on object oriented programming, has become a classic that should not be missing from the bookshelf of any developer worth their salt, even if it is just to criticize it.



Cartoon from osnews.com about code quality.

There are a lot of definitions for the term Clean Code, but personally I prefer my friend Carlos Blé's definition, since it goes very well with the objectives of this book.

“Clean Code is one that has been written with the intention of making it understandable for someone else (or yourself in the future).” – Carlos Blé

Developers often write code without the explicit intention that it will be understood by someone else, since most of the time we simply focus on implementing a solution that works and solves the problem.

Trying to understand another person's code, or even code that we ourselves wrote some weeks ago, can become really difficult. That is why making an extra effort to produce readable and intuitive solutions is the basis for reducing the maintenance costs of the software we write.

Next we will look at some of the sections of Uncle Bob's book that deal mostly with code readability. If you know the book or have read it, you will see that I have added some concepts and discarded others, as well as including simple examples applying to JavaScript.

Variables, names and scope

“Our code has to be simple and direct, it should read as easily as a well written text.” – Grady Booch

Reading our code should be like reading a well written text, and for this reason we should always try to choose good names, try to use the declaration of variables correctly and understand that the concept of scope is fundamental in JavaScript.

The names of variables, functions or methods and classes must be selected carefully so that they give expression and meaning to our code. In this chapter, apart from going more deeply into some important details related to variables and their scope, we will also see some guidelines and examples to help in trying to get better at choosing good names.

Correct use of *var*, *let* y *const*

In classic JavaScript, before ES6, we only had one way of declaring variables and this was by using the keyword `var`. `let` and `const` was introduced with ES6, so since then we have had three keywords for declaring variables.

It would be ideal to be able to avoid the use of `var`, because it doesn't allow the definition of variables with block scope, so it might result in unexpected and not very intuitive behaviour. This does not happen with the variables defined by `let` and `const`, which allow the declaration of this kind of scope, as we will see at the end of this chapter.

The difference between `let` and `const` is that the value of `const` cannot be reassigned, although it can be modified. In other words, the value could be modified (mutated) in the case of an object, but not if it is a primitive type. This is why we should use the keyword `const` in the variables whose value we don't mean to change. This could help us to improve the intention of our code.

Var usage example

```
1 var variable = 5;
2 {
3   console.log('inside', variable); // 5
4   var variable = 10;
5 }
6
7 console.log('outside', variable); // 10
8 variable = variable * 2;
9 console.log('changed', variable); // 20
```

You can access the interactive example [from here](#)⁴

```
1 let variable = 5;
2
3 {
4   console.log('inside', variable); // error
5   let variable = 10;
6 }
7
8 console.log('outside', variable); // 5
9 variable = variable * 2;
10 console.log('changed', variable); // 10
```

You can access the interactive example [from here](#)⁵

```
1 const variable = 5;
2 variable = variable * 2; // error
3 console.log('changed', variable); // doesn't get here
```

You can access the interactive example [from here](#)⁶

⁴<https://repl.it/@SoftwareCrafter/CLEAN-CODE-var>

⁵<https://repl.it/@SoftwareCrafter/CLEAN-CODE-let>

⁶<https://repl.it/@SoftwareCrafter/CLEAN-CODE-const>

Pronounceable and expressive names

The names of the variables, essential in English, must be pronounceable. This means that they should not be abbreviations or have an underscore or middle hyphen, prioritizing the CamelCase style. Also, we should try not to save characters in the names, the idea is that they are as expressive as possible.

```
1 //bad
2 const yyymmddstr = moment().format('YYYY/MM/DD');
3
4 //better
5 const currentDate = moment().format('YYYY/MM/DD');
```

Absence of technical information in the names

If we are building a vertical software (business oriented), we should try not to include technical information in the names. In other words, we should avoid including information related to the technology, such as the data type or the Hungarian notation, the class type, etc. This is allowed in horizontal software development or general purpose libraries.

```
1 //bad
2 class AbstractUser(){...}
3
4 //better
5 class User(){...}
```

Establish a ubiquitous language

The term “ubiquitous language” was introduced by Eric Evans in his famous book on DDD, *Implementing Domain-Driven Design*, also known as “the red book”. Although

DDD is outside of the subject matter of this book, I believe that making use of ubiquitous language is tremendously important in obtaining a coherent lexicon.

Ubiquitous language is a process in which it is necessary to establish a common language between programmers and stakeholders (domain experts), based on the definitions and terminology used in the business.

A good way to start this process might be to create a glossary of terms. This will allow us, on the one hand, to improve communication with business experts, and on the other, to help us choose more precise names to maintain a homogeneous nomenclature throughout the application.

We should also use the same vocabulary to refer to each concept, we must not use “Client” in some places, “Customer” in others, and “User” in others, unless they clearly represent different concepts.

```
1 //bad
2 getUserInfo();
3 getClientData();
4 getCustomerRecord();
5
6 //better
7 getUser()
```

Names by data type

Arrays

Arrays are an iterable list of elements, generally of the same type. That is why pluralizing the variable name might be a good idea:

```
1 //bad
2 const fruit = ['apple', 'banana', 'strawberry'];
3 // regular
4 const fruitList = ['apple', 'banana', 'strawberry'];

5 // good
6 const fruits = ['apple', 'banana', 'strawberry'];
7 // better
8 const fruitNames = ['apple', 'banana', 'strawberry'];
```

Booleans

Booleans can only have two values: true and false. For this reason, using a prefix such as “is”, “has” or “can” will help us to identify the variable type, which will improve the legibility of our code.

```
1 //bad
2 const open = true;
3 const write = true;
4 const fruit = true;
5
6 // good
7 const isOpen = true;
8 const canWrite = true;
9 const hasFruit = true;
```

Numbers

For numbers it is worthwhile choosing words that describe numbers, such as “min”, “max” or “total”:

```
1 //bad
2 const fruits = 3;
3
4 //better
5 const maxFruits = 5;
6 const minFruits = 1;
7 const totalFruits = 3;
```

Functions

Names of functions should represent actions, so they should be made by using the verb that represents the action, followed by a noun. These names should be descriptive and concise. This means that the name of the function should express what it does, but this must also be abstracted from the implementation of the function.

```
1 //bad
2 createUserIfNotExists()
3 updateUserIfNotEmpty()
4 sendEmailIfFieldsValid()
5
6 //better
7 createUser(...)
8 updateUser(...)
9 sendEmail()
```

For access, modify, or predicate functions, the name should be prefixed with “get”, “set”, and “is”, respectively.

```
1 getUser()
2 setUser(...)
3 isValidUser()
```

Get & set

In the case of getters and setters, it would be interesting to use the keywords; “get” and “set” when we are accessing properties of objects. These were introduced in ES6 and allow us to define accessor methods:

```
1  class Person {
2    constructor(name) {
3      this._name = name;
4    }
5
6    get name() {
7      return this._name;
8    }
9
10   set name(newName) {
11     this._name = newName;
12   }
13 }
14
15 let person = new Person('Miguel');
16 console.log(person.name); // Outputs 'Miguel'
```

Classes

Classes and objects should have names made up of a noun or noun phrases such as User, UserProfile, Account, or AddressParser. We should avoid generic names such as Manager, Processor, Data or Info.

These names must be chosen carefully, as they are the first step in defining the responsibility of the class. If we choose names that are too generic we tend to create classes with multiple responsibilities.

Scope of the variables

In addition to writing appropriate names for variables, understanding how their scope works in JavaScript is critical. The scope refers to the visibility and useful

life of a variable. The scope, in essence, determines where in our program we have access to a certain variable.

There are three main types of scope in JavaScript: the global scope, the local or function scope, and the block scope.

Global Scope

Any variable that is not within a block of a function will be within the global scope. These variables will be accessible from any part of the application:

```
1  let greeting = 'hello world!';
2
3  function greet(){
4      console.log(greeting);
5  }
6
7  greet(); // "Hello world!"
```

Block Scope

Blocks in Javascript are delimited by braces, an opening one '{', and a closing one '}'. As mentioned in the section on "Correct use of var, let and const", to define variables with block scope we must use let or const:

```
1  {
2      let greeting = "Hello world!";
3      var lang = "English";
4      console.log(greeting); //Hello world!
5  }
6
7  console.log(lang); // "English"
8  console.log(greeting); // Uncaught ReferenceError: greeting is not def\
9  ined
```

In this example, it is clear that the variables defined with `var` can be used outside the block, since these types of variables are not encapsulated within the blocks. For this reason, and in accordance with the aforementioned, we should avoid their use so as not to encounter unexpected behaviors.

Static vs. dynamic scope

The behaviour of the scope of variables in JavaScript is static by nature. This means that it is determined at compile time rather than at run time. This is also often referred to as a lexical scope. Let's see an example:

```
1  const number = 10;
2  function printNumber() {
3      console.log(number);
4  }
5
6  function app() {
7      const number = 5;
8      printNumber();
9  }
10
11 app(); //10
```

In this example, `console.log (number)` will always print the number 10 no matter where the `printNumber ()` function is called from. If JavaScript were a dynamically scoped language, `console.log (number)` would print a different value depending on where the `printNumber ()` function runs.

Hoisting

In JavaScript the declarations of variables and functions are allocated in the memory at compile time; on a practical level it is as if the interpreter had moved these declarations to the beginning of their scope. This behavior is known as hoisting.

Thanks to hoisting we can execute a function before its declaration:

```
1 greet(); //"Hello world";  
2 function greet(){  
3     let greeting = 'Hello world!';  
4     console.log(greeting);  
5 }
```

By assigning the declaration in the memory, it is as if you “raised” the function to the beginning of its scope:

```
1 function greet(){  
2     let greeting = 'Hello world!';  
3     console.log(greeting);  
4 }  
5 greet(); //"Hello world";
```

In the case of variables, hoisting can generate unexpected behaviors, since, as we have said, it only applies to the declaration and not to its assignment:

```
1 var greet = "Hi";  
2 (function () {  
3     console.log(greet);// "undefined"  
4     var greet = "Hello";  
5     console.log(greet); //"Hello"  
6 })();
```

In the first console.log of this example, you are expected to write “Hi”, but as mentioned earlier, the interpreter “raises” the variable declaration to the top of its scope. Therefore, the behavior of the previous example would be equivalent to writing the following code:

```
1 var greet = "Hi";
2 (function () {
3     var greet;
4     console.log(greet); // "undefined"
5     greet = "Hello";
6     console.log(greet); // "Hello"
7 })();
```

I have used this example because I think it is very illustrative in explaining the concept of hoisting, but re-declaring a variable with the same name and also using var to define them is a very bad idea.

Functions

“You know you are working on Clean Code when each function you read turns out to be pretty much what you expected.” - Ward Cunningham

Functions are the most basic organizational entity in any program. Therefore, they should be easy to read and understand, as well as clearly transmitting their intention. Before looking more deeply into what they should be like, we'll explore the different ways they can be defined: declaration, expressions, and arrow functions. We will also explain how the 'this' object works in arrow functions, and we will see that they can be a bit tricky in JavaScript.

Function declaration

The classic way to define functions in JavaScript is through the declaration of functions. They are declared with the function keyword followed by the function name and parentheses. It may or may not have parameters. Then, between braces, we will have the set of instructions and optionally the return keyword and the return value.

```
1  function doSomething(){  
2      return "Doing something";  
3  }  
4  
5  doSomething() //"Doing something"
```

Function Expression

An expression of a function has a syntax similar to the declaration of a function, except that we assign the function to a variable:

```
1  const doSomething = function(){
2      return "Doing something";
3  }
4
5  doSomething() //"Doing something"
```

Function expression with the arrow function

With the arrival of ES6, the syntax of the arrow function was added to the language, which is a way of defining functions that is much more readable and concise.

```
1  const doSomething =() => "Doing something";
2  //The return is implicit if we don't add the braces.
```

Arrow functions are ideal for declaring lambda expressions (inline functions), since noise in the syntax is reduced and the expressiveness and intentionality of the code is improved.

```
1  //without arrow functions
2  [1, 2, 3].map(function(n){return n * 2})
3  //with arrow functions
4  [1,2,3].map(n => n * 2)
```

Arrow functions are also very useful when writing curried functions. This is a function that takes one argument, returns a function that takes the next argument, and so on. With arrow functions, this process can be shortened, resulting in much more readable code.

```
1  //without arrow functions
2  function add(x){
3      return function(y){
4          return x + y;
5      }
6  }
7  //with arrow functions
8  const add = x => y => x + y;
9
10 const addTwo = add(2);
11 console.log(addTwo(5))//7
```

How the 'this' object works in arrow functions

Another interesting feature of arrow functions is that they change the default behavior of the 'this' object in JavaScript. When we create an arrow function, its 'this' value is permanently associated with the 'this' value of its immediate outer scope; 'window' in the case of the global scope of the browser or 'global' in the case of the global scope of NodeJS:

```
1  const isWindow = () => this === window;
2  isWindow(); // true
```

When we are in the local scope of a method, the value of 'this' would be the value of the scope of the function. Let's see an example:


```
1 const counter = {
2   number: 0,
3   increase() {
4     setInterval(() => ++this.number, 1000);
5   }
6 };
7
8 counter.increase(); //1 2 3 4 5
```

Inside the arrow function, the value of “this” is the same as in the increase() method. Although this may seem like the expected behavior, it wouldn’t be the case if we didn’t use arrow functions. Let’s look at the same example using a function created with the function keyword:

this in arrow functions

```
1 const counter = {
2   number: 0,
3   increase() {
4     setInterval(function() { ++this.number }, 1000);
5   }
6 };
7
8 counter.increase(); //NaN NaN NaN ...
```

Although NaN (not a number) is not the intuitive result, it makes sense in JavaScript, since inside setInterval () “this” has lost the reference to the counter object. Before the advent of arrow functions, this problem with callbacks used to be corrected by making a copy of the “this” object:

```
1 const counter = {
2     number: 0,
3     increase() {
4         const that = this;
5
6         setInterval(function(){ ++that.number}, 1000);
7     };
8
9 counter.increase(); //1 2 3 ...
```

Or, by binding the “this” object, using the bind function:

this in arrow functions

```
1 const counter = {
2     number: 0,
3     increase() {
4         setInterval(function() { ++this.number }.bind(this), 1000);
5     }
6 };
7
8 counter.increase();
```

In this example it is shown that both solutions generate a lot of noise, so using the arrow functions in the callbacks in which “this” is used becomes essential.

Immediately-invoked Function Expressions (IIFE)

One way to define lesser known functions is through IIFE. An IIFE (Immediately-Invoked Function Expressions) is a function that is executed when defined:

```
1 (function(){  
2     // ... do something  
3 })()
```

This pattern was widely used for creating a block scope before `let` and `const` were introduced. Since the arrival of ES6 this does not make much sense, but it is interesting to know:

```
1 (function() {  
2     var number = 42;  
3 }());  
4  
5 console.log(number); // ReferenceError
```

As we have seen in the block scope section, using `let` and `const` to define it is much more intuitive and concise:

```
1 {  
2     let number = 42;  
3 }  
4  
5 console.log(number); // ReferenceError
```

Parameters and arguments

The arguments are the values with which we call the functions, while the parameters are the named variables that receive these values within our function:

```
1 const double = x => x * 2; // x is the parameter of our function  
2 double (2); // 2 is the argument with which we call our function
```

Limit the number of arguments

An important recommendation is to limit the number of arguments a function receives. In general terms we should limit our functions to a maximum of three parameters. In the case of having to exceed this number, it might be a good idea to add one more level of indirection through an object:

```
1  function createMenu (title, body, buttonText, cancellable) {
2    // ...
3  }
4
5  function createMenu ({title, body, buttonText, cancellable}) {
6    // ...
7  }
8
9  createMenu ({
10    title: 'Foo',
11    body: 'Bar',
12    buttonText: 'Baz',
13    cancellable: true
14  });
```

Generally, as we saw in the chapter on variables and names, we should avoid making use of abbreviations, except when dealing with lambda expressions (inline functions), since their scope is very small and readability problems would not appear:

```
1  const numbers = [1, 2, 3, 4, 5];
2  numbers.map (n => n * 2); // The argument to map is a lambda expression.
```

Default parameters

Since ES6, JavaScript allows function parameters to be initialized with default values.

```
1 // With ES6
2 function greet (text = 'world') {
3     console.log ('Hello' + text);
4 }
5
6 greet (); // no parameter. Hello world
7 greet (undefined); // undefined. Hello world
8 greet ('crafter') // with parameter. Hello crafter
```

In classic JavaScript, to do something as simple as this we had to check if the value is undefined and assign it the desired default value:

```
1 // Before ES6
2 function greet (text) {
3     if (typeof text === 'undefined')
4         text = 'world';
5
6     console.log ('Hello' + text);
7 }
```

Although we should not abuse the default parameters, this syntax can help us to be more concise in some contexts.

Rest parameter and spread operator

The operator ... (three dots) is known as the rest parameter or as the spread operator, depending on where it is used.

The rest parameter unifies the remaining arguments in a function call when the number of arguments exceeds the number of parameters declared in it.

In a way, the rest parameter works in the opposite way to spread: while spread expands the elements of an array or object, rest unifies a set of elements in an array.

```
1  function add (x, y) {
2    return x + y;
3  }
4
5  add (1, 2, 3, 4, 5) // 3
6
7  function add (... args) {
8    return args.reduce ((previous, current) => previous + current, 0)
9  }
10
11 add (1, 2, 3, 4, 5) // 15
12
13 // The “rest” parameter is the last parameter of the function and, as w\
14 e have mentioned, it is an array:
15
16 function process (x, y, ... args) {
17   console.log (args)
18 }
19 process (1, 2, 3, 4, 5); // [3, 4, 5]
```

Like the default parameters, this feature was introduced in ES6. In order to access the additional arguments in classic JavaScript, we have the arguments object:

```
1  function process (x, y) {
2    console.log (arguments)
3  }
4
5  process (1, 2, 3, 4, 5); // [1, 2, 3, 4, 5]
```

The “arguments” object presents some problems. The first one, is that although it looks like an array, it is not an array, and therefore does not implement the functions of array.prototype. Also, unlike “rest”, it can be overwritten and it contains all the arguments, not just the remaining ones. That’s why its use should usually be avoided.

On the other hand, the spread operator splits an object or an array into multiple individual elements. This allows you to expand expressions in situations where multiple values are expected, such as in function calls or in array literals:

```
1  function doStuff (x, y, z) {}
2  const args = [0, 1, 2];
3  // with spread
4  doStuff (... args);

5
6  // no spread
7  doStuff.apply (null, args);
8
9  // spread in math functions
10 const numbers = [9, 4, 7, 1];
11 Math.min (... numbers); // 1
```

Spread also allows us to clone objects and arrays in a very simple and expressive way:

```
1  const post = {title: "Spread operator", content: "lorem ipsum ..."}
2  // cloned with Object.assign ()
3  const postCloned = Object.assign ({}, post);
4  // cloned with the spread operator
5  const postCloned = {... post};
6
7  const myArray = [1, 2, 3];
8  // cloned with slice ()
9  const myArrayCloned = myArray.slice ();
10 // cloned with the spread operator ()
11 const myArrayCloned = [... myArray];
```

We can also use the spread operator to concatenate arrays:

```
1  const arrayOne = [1, 2, 3];
2  const arrayTwo = [4, 5, 6];
3
4  // concatenation with concat ()
5
6  const myArray = arrayOne.concat (arrayTwo); // [1, 2, 3, 4, 5, 6]
7  // concatenation with spread operator
8  const myArray = [... arrayOne, ... ArrayTwo]; // [1, 2, 3, 4, 5, 6]
```

Indentation size and levels

Simplicity is a fundamental pillar when it comes to writing good code and, therefore, one of the key recommendations is that our functions need to be small.

Defining an exact number is tricky: I sometimes write single-line functions, although they usually have 4 or 5 lines. This does not mean that you never write longer functions, for example 15 or 20 lines. However, when I reach this length I try to analyze whether it can be divided into several functions.

If your functions, as a general rule, tend to be too long or have too many levels of indentation, they are likely to do too much. This brings us to another recommendation, perhaps the most important: functions should do one thing and do it well.

Another fundamental point to keep our functions simple is to try to limit the indentation levels to 1 or 2. To do this, we must avoid nesting conditionals and loops. This will allow us to keep the spaghetti code at bay and also reduce the cyclomatic complexity of the function.

Let's see an example of how our functions should NOT be:


```
1  const getPayAmount = () => {
2    let result;
3    if (isDead){
4      result = deadAmount();

5    }
6    else {
7      if (isSeparated){
8        result = separatedAmount();
9      }
10     else {
11       if (isRetired){
12         result = retiredAmount();
13       }
14       else{
15         result = normalPayAmount();
16       }
17     }
18   }
19   return result;
20 }
```

Guard clauses

Guard clauses, also known as assertions or preconditions, are pieces of code that check for a series of conditions before going ahead with executing the function.

In the example above, as you can see, we have too many nested conditionals. To solve it we could replace the edge cases with guard clauses:

Guard clauses

```
1  const getPayAmount = ( ) => {  
2    if (isDead)  
3      return deadAmount();  
  
4    if (isSeparated)  
5      return separatedAmount();  
6  
7    if (isRetired)  
8      return retiredAmount();  
9  
10   return normalPayAmount();  
11  
12 }
```

Avoid using the 'else' keyword

Another strategy I use to avoid nesting is to not use the 'else' keyword. In my projects I try to avoid it whenever possible; in fact, I've worked on projects with thousands of lines of code without using "else" at all. To achieve this, I usually prioritize declarative style, make use of guard clauses when using conditional structures, or replace "if/else" structures with the ternary operator, which results in much more understandable and expressive code:

```
1  //if/else  
2  const isRunning = true;  
3  if(isRunning){  
4    stop()  
5  }  
6  else{  
7    run()  
8  }  
9  //ternary operator  
10 isRunning ? stop() : run()
```

When you use this operator, you should try to keep the expression as simple as possible, otherwise, it might become difficult to read.

Prioritize assertive conditions

Although this section is not directly related to indentation levels, I think it is interesting to mention it, since it helps us improve the readability of conditionals.

The evidence tells us that affirmative sentences are usually easier to understand than negative ones, for this reason we should reverse, whenever possible, the negative conditions to make them affirmative:

```
1 //Negative
2 if(!canNotFormat){
3     format()
4 }
5
6 //Positive
7 if(canFormat){
8     format()
9 }
```

Declarative vs imperative style

Although JavaScript is not a pure functional language, it does offer us some elements of functional programming that allow us to write a much more declarative code.

A good practice might be to prioritize the high-level “map”, “filter”, and “reduce” functions over the control and conditional structures. This, in addition to favoring the composition, will allow us to obtain much more expressive and smaller functions.

```
1  const orders = [  
2    { productTitle: "Product 1", amount: 10 },  
3    { productTitle: "Product 2", amount: 30 },  
4    { productTitle: "Product 3", amount: 20 },  
  
5  ]; { productTitle: "Product 4", amount: 60 }  
6  
7  
8  //worse  
9  function imperative(){  
10   let totalAmount = 0;  
11  
12   for (let i = 0; i < orders.length; i++) {  
13     totalAmount += orders[i].amount;  
14   }  
15  
16   console.log(totalAmount); // 120  
17 }  
18 //better  
19 function declarative(){  
20   function sumAmount(currentAmount, order){  
21     return currentAmount + order.amount;  
22   }  
23  
24   function getTotalAmount(orders) {  
25     return orders.reduce(sumAmount, 0);  
26   }  
27  
28   console.log(getTotalAmount(orders)); // 120  
29 }  
30  
31 imperative();  
32 declarative();
```

You can access the interactive example [here](https://repl.it/@SoftwareCrafter/CLEAN-CODE-declarativo-vs-imperativo)⁷.

⁷<https://repl.it/@SoftwareCrafter/CLEAN-CODE-declarativo-vs-imperativo>

Anonymous functions

We saw in the names section how the value of a good name is critical for readability. When we choose a bad name, it has the opposite effect, so sometimes the best way to choose good names is to not have to do it. This is where the strength of anonymous functions comes in and so you should use them where context allows. This will prevent aliases and bad names from spreading through your code. Let's see an example:

```
1 function main(){
2   const stuffList = [
3     { isEnabled: true, name: 'justin' },
4     { isEnabled: false, name: 'lauren' },
5     { isEnabled: false, name: 'max' },
6   ];
7
8   const filteredStuff = stuffList.filter(stuff => !stuff.isEnabled);
9   console.log(filteredStuff);
10 }
11
12 main();
```

The function `stuff => !stuff.isEnabled` is a very simple predicate and it would make no sense to extract it. You can access the full example [here](https://repl.it/@SoftwareCrafter/CLEAN-CODE-funciones-anonimas)⁸.

Referential transparency

Many times we come across features that promise to do one thing but actually have hidden side effects. We must try to avoid this as much as possible, so it is usually a good idea to apply the principle of referential transparency on our functions.

A function fulfills the principle of referential transparency if, for an input value, it always produces the same output value. These types of functions are also known as pure functions and are the basis of functional programming.

⁸<https://repl.it/@SoftwareCrafter/CLEAN-CODE-funciones-anonimas>

```
1  //bad
2  function withoutReferentialTransparency(){
3      let counter = 1;
4
5      function increaseCounter(value) {
6          counter = value + 1;
7      }
8
9      increaseCounter(counter);
10     console.log(counter); // 2
11 }
12
13 //better
14 function withReferentialTransparency(){
15     let counter = 1;
16
17     function increaseCounter(value) {
18         return value + 1;
19     }
20
21     console.log(increaseCounter(counter)); // 2
22     console.log(counter); // 1
23 }
24
25 withoutReferentialTransparency();
26 withReferentialTransparency();
```

You can access the full example [from here](https://repl.it/@SoftwareCrafter/CLEAN-CODE-transparencia-referencial)⁹

DRY Principle

Considering that code duplication is often the root cause of multiple problems, a good practice would be to implement DRY (don't repeat yourself). This principle

⁹<https://repl.it/@SoftwareCrafter/CLEAN-CODE-transparencia-referencial>

will avoid multiple headaches such as having to test the same thing several times, as well as helping to minimize the amount of code to maintain.

The ideal way of doing this would be to extract the duplicate code to a class or function and use it where we need it. Many times this duplication will not be so evident and it will be our experience that helps us to detect it, do not be afraid to refactor when you detect these situations.

DRY Principle

```
1  const reportData = {
2    name: "Software Crafters",
3    createdAt: new Date(),
4    purchases: 100,
5    conversionRate: 10,
6  }

7
8  function withoutDRY(){
9    function showReport(reportData) {
10      const reportFormatted = `
11        Name: ${reportData.name}
12        Created at: ${reportData.createdAt}
13        Purchases: ${reportData.purchases}
14        Conversion Rate: ${reportData.conversionRate}%`
15      console.log("Showing report", reportFormatted)
16    }

17
18    function saveReport(reportData) {
19      const reportFormatted = `
20        Name: ${reportData.name}
21        Created at: ${reportData.createdAt}
22        Purchases: ${reportData.purchases}
23        Conversion Rate: ${reportData.conversionRate}%`
24      console.log("Saving report...", reportFormatted)
25    }
26
27    showReport(reportData)
28    saveReport(reportData)
```

```
29  }
30
31  function withDRY(){
32    function formatReport(reportData){
33
34      return `
35        Name: ${reportData.name}
36        Created at: ${reportData.createdAt}
37        Purchases: ${reportData.purchases}
38        Conversion Rate: ${reportData.conversionRate}%`
39    }
40
41    function showReport(reportData) {
42      console.log("Showing report...", formatReport(reportData));
43    }
44
45    function saveReport(reportData) {
46      console.log("Saving report...", formatReport(reportData));
47    }
48
49    showReport(reportData)
50    saveReport(reportData)
51  }
```

You can access a full example from [here](https://repl.it/@SoftwareCrafter/CLEAN-CODE-principio-DRY)¹⁰.

Command-Query Separation (CQS)

The Command-Query Separation design principle was first introduced by Bertrand Meyer in his book Object Oriented Software Construction. The fundamental idea behind this principle is that we should try to divide the functions of a system into two clearly separate categories:

- **Queries:** are pure functions that respect the principle of referential transparency, that is, they return a value and do not alter the state of the system.

¹⁰<https://repl.it/@SoftwareCrafter/CLEAN-CODE-principio-DRY>

They always return a value.

- **Commands:** these are functions that change the intrinsic state of the system, that is, they generate a side effect. They can also be known as modifiers or mutators. They should not return any value (void type).

Let's take a look at the signature of the following interface:

```
1 interface UserRepository
2 {
3     Create(user:User): void;
4     GetByEmail(email:string):User;
5     GetAllByName(string name): User[]
6 }
```

As you can see, the Create method does not return any value, its only action is to create a new user, it mutates the system state and is therefore a command.

On the other hand, the GetByEmail and GetAllByName functions are queries that return one user by email or several users by name, respectively. If they are well designed, they should not generate any side effects, that is, they should not change the state of the system.

The main value of this principle is that separating commands from queries can be extremely useful. This will allow us to reuse and compose the queries in different parts of the code where needed. As a consequence, we get a more robust code, free of duplications.

Efficient algorithms

Bjarne Stroustrup, inventor of C++ and author of several books, understands the concept of clean code as code that is elegant and efficient. In other words, it is not only a pleasure to read, but it also performs well. But how do we know if our code is performing properly? Well, for this we must know the *Big O* and how it classifies the algorithms that we have encoded.

Big-O notation

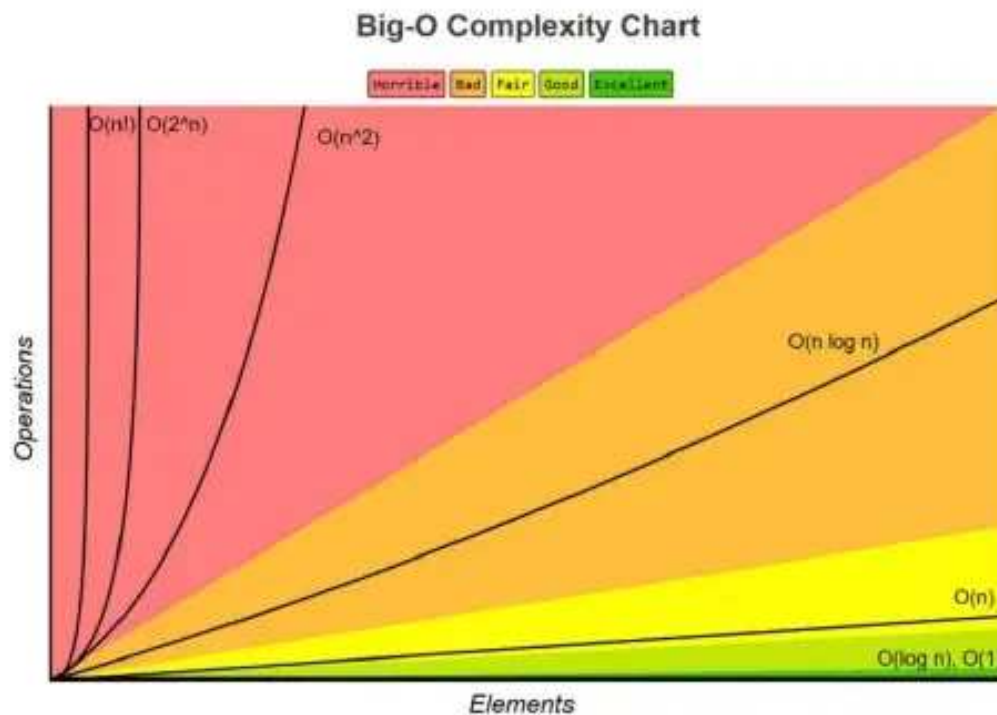
The Big-O notation, also known as asymptotic notation or Landau notation (in honor of one of its inventors, Edmund Landau), is used to measure the performance or complexity of an algorithm.

In essence it is a mathematical approach that helps us describe the behavior of an algorithm, both temporal and spatial. That is, how long it will take to run or how much memory it will occupy while running, based on the number of items that need to be processed.

For example, if the execution time of an algorithm grows linearly with the number of elements, we will say that the algorithm is of complexity $O(n)$. On the other hand, if the algorithm is independent of the amount of data that is going to be processed, we will be facing an algorithm of complexity $O(1)$. Here below, we can see the most common big-O notations, ordered from least to greatest complexity, along with some examples.

- **$O(1)$ constant:** the operation does not depend on the size of the data. For example, accessing an element of an array. Logarithmic $O(\log n)$: logarithmic complexity occurs in cases where it is not necessary to go through all the elements. For example, the binary search algorithm in an ordered list or traversing a binary tree.
- **Linear $O(n)$:** the execution time is directly proportional to the size of the data. It grows in a straight line. As an example, any algorithm that makes use of a simple loop is valid, such as a sequential search.
- **$O(n \log n)$:** is a bit worse than linear, but not much. It is applied in the case of sort algorithms such as Quicksort or Heapsort.
- **$O(n^2)$ quadratic:** is typical of algorithms that need to iterate through all the elements in each of the elements it needs to process. For example, any algorithm that makes use of two nested loops, such as the search algorithm Bubble Sort. In the case of adding another loop, the algorithm would become cubic in complexity.
- **$O(2^n)$ exponential:** These are functions that double in complexity with each element added to the processing. It usually occurs in multiple recursive calls. An example is the calculation of the Fibonacci series recursively. We will develop the Fibonacci algorithm with TDD at the end of the book.

- **$O(n!)$ Combinatorial explosion:** these are algorithms that cannot be solved in polynomial time, also known as NP (nondeterministic polynomial time). A typical example is the traveler salesman problem.



Big-O Notation

As you can see in the previous graph, quadratic complexity can cause the algorithms to become too slow when handling large amounts of data. Sometimes this can be a trade off that puts us in the position of choosing between a more elegant or a more efficient design. In these cases it is usually a good idea not to rush into premature optimization, but that all depends on the context.

Classes

“If you want to be a productive programmer, make the effort to write readable code.” – Robert C. Martin

A class, in addition to being an abstraction through which we represent entities or concepts, is a very powerful organizational element. That is why we should try to pay special attention when designing them. Before delving into how they should be designed, let's look at some characteristics of classes and objects in JavaScript.

JavaScript is a prototype-based object-oriented language, rather than class-based as such. In the ES6 version of the language, the keyword “class” was introduced to define classes following the same pattern of classic OOP languages such as Java or C#. Actually, this class syntax does not really offer additional functionality, it simply provides a cleaner and more elegant style compared to the direct use of constructor functions and/or the prototype chain.

Prototype and modern ECMAScript

All JavaScript objects link to a prototype object from which they inherit all their properties. Prototypes allow many of the classic object-oriented design techniques to be integrated, but before ES6, it was done through a messy and complex mechanism. Now, we will compare how some elements of OOP were implemented before and after ES6.

Constructors and constructor functions

As you know, in OOP a constructor is a function that allows us to initialize an object of a specific class. Before ES6 we didn't have a formal class notation and we had to use constructor functions for this purpose. The only peculiarity of these types of function is that they use the *this* keyword to define the properties that the object will have, and they are initialized with the *new* keyword:

```
1 // Before ES6
2 function Person(name) {
3     this.name = name;
4 }
5
6 var person = new Person("Miguel");
7 console.log(person.name); // 'Miguel'
```

Since the introduction of ES6, we have been able to use the *class* keyword to define “classes” as we do in other object-oriented languages, although internally JavaScript still uses prototypes. Let’s see what the previous example would look like with this syntax:

Constructors

```
1 //After ES6
2 class Person{
3     constructor(name){
4         this.name = name;
5     }
6 }
7
8 const person = new Person("miguel");
9 console.log(person.name); // 'Miguel'
```

As you can see, this syntax is much more intuitive than the previous one. Currently, making use of constructor functions in modern JavaScript projects does not make any sense as it reduces the readability of the project.

Methods

Methods represent operations that can be performed with objects of a particular class. Before ES6, in order to define them, we had to assign them directly to the *prototype* object after declaring the constructor function:

```
1 // Before ES6
2 function Person(name) {
3     this.name = name;
4 }
5
6 Person.prototype.greet = function(){
7     return "Hi, I am " + this.name;
8 }
9
10 var person = new Person("Miguel");
11 console.log(person.greet()); // 'Hi, I am Miguel'
```

The ES6 class syntax allows us to do this in a more readable and cohesive way:

Methods

```
1 class Person{
2     constructor(name){
3         this.name = name;
4     }
5
6     greet(){
7         return `Hi, I am ${this.name}`;
8     }
9 }
10
11 const person = new Person("miguel");
12 console.log(person.greet()); // 'Hi, I am Miguel'
```

Inheritance and prototype chain

Inheritance is a typical OOP technique that allows us to define new classes based on existing ones in order to reuse code. However, as we will see in this chapter, inheritance is not the best option for code reuse, although there are certain contexts in which it can be applied very well.

Let's see how inheritance is implemented using prototypes with the traditional ES5 syntax. To do this, we will create a programmer object that inherits from the person object that we created in the previous examples: (and yes, in JavaScript it is more accurate to speak of “inheritance between objects” than of “classes”):

```
1  // ES5
2  function Programmer(name) {
3      Person.call(this, name);
4  }
5
6  Programmer.prototype = Object.create(Person.prototype);
7
8  Programmer.prototype.writeCode = function(coffee) {
9      if(coffee)
10         console.log( 'I am programming');
11     else
12         console.log('I can not, I do not have coffee');
13 };
14
15 var programmer = new Programmer("Miguel");
16 programmer.greet(); // 'I am Miguel'
17 programmer.writeCode(); // 'I can not, I do not have coffee'
```

As you can see, we first define a new constructor function called *Programmer*. Then, we assign to its prototype a new object based on the *Person* object prototype, this allows us to inherit all the functionality implemented in the *Person* object. Finally, we define the *writeCode (coffee)* method in the same way that we did in the previous example.

It is clear that the direct use of prototype is not intuitive at all. Let's see how the same example looks with the class syntax:

```
1 // ES6
2 class Programmer extends Person{
3     constructor(name){
4         super(name);
5     }
6
7     writeCode(coffee){
8         coffee ? console.log( 'Hi, I am Miguel' ) : console.log('I can not, \
9 I do not have coffee. ');
10    }
11 }
12
13 const programmer = new Programmer("Miguel");
14 programmer.greet(); // 'Hi, I am Miguel'
15 programmer.writeCode(); // 'I can not, I do not have coffee'
```

Class syntax allows you to write more readable and intuitive code. And by the way, in the class constructor we are passing the *name* parameter to the parent class using the `super` keyword. This type of practice should be minimized since the rigidity and coupling of our code are increased.

Reduced size

The classes, as we saw in the functions, should be reduced in size. To achieve this we should start by **choosing a good name**. A proper name is the first way to limit the size of a class, as it should describe the responsibility that the class has.

Another guideline that helps us to keep our classes to an adequate size is to try to apply **the single responsibility principle**. This principle says that a class must not have more than one responsibility, that is, it must not have more than one reason for being modified (we will expand this definition in the SOLID principles section). Let's see an example:


```
1  class UserSettings {
2      private user: User;
3      private settings: Settings;
4
5      constructor(user) {
6          this.user = user;
7      }
8
9      changeSettings(settings) {
10         if (this.verifyCredentials()) {
11             // ...
12         }
13     }
14
15     verifyCredentials() {
16         // ...
17     }
18 }
```

The *UserSettings* class has two responsibilities: on the one hand, it has to manage the user's settings and it is also in charge of handling the credentials. In this case, it could be interesting to extract the verification of the credentials to another class, for example *UserAuth*, and have this class be responsible for managing the operations related to the handling of the credentials. We would only have to inject it through the *UserSettings* class constructor and use it where we need it, in this case in the *changeSettings* method.

```
1  class UserAuth{
2      private user: User;
3
4      constructor(user: User){
5
6          this.user = user
7
8      verifyCredentials(){
9          //...
10     }
11 }
12
13 class UserSettings {
14     private user: User;
15     private settings: Settings;
16     private auth: UserAuth;
17
18     constructor(user: User, auth:UserAuth) {
19         this.user = user;
20         this.auth = auth;
21     }
22
23     changeSettings(settings) {
24         if (this.auth.verifyCredentials()) {
25             // ...
26         }
27     }
28 }
```

This way of designing classes allows us to keep responsibilities well defined, as well as containing their size. We will delve into this in the chapter on the single responsibility principle.

Organization

Classes should start with a list of variables. If there are public constants, they should appear first. After that, come the private static variables and then the private instance variables; if we use public instance variables, these should be last.

Public methods or functions should follow the list of variables. To do this we will start with the constructor method. If a named constructor is used, it would go first and then the private constructor method would follow that. Then we will put the static functions of the class and, if you have related private methods, they go next. Then the rest of the instance methods would go in order of importance from the highest to the lowest, leaving the accessors (getters and setters) to the end.

For this example we will use a small class built with Typescript, since it makes it easier for us to set methods and private variables.

```
1  class Post {
2      private title : string;
3      private content: number;
4      private createdAt: number;
5
6      static create(title:string; content:string){
7          return new Post(title, content)
8      }
9
10     private constructor(title:string; content:string){
11         this.setTitle(title);
12         this.setContent(content);
13         this.createdAt = Date.now();
14     }
15
16     setTitle(title:string){
17         if(StringUtils.isNullOrEmpty(title))
18             throw new Error('Title cannot be empty')
19
20         this.title = title;
```

```
21     }
22
23     setContent(content:string){
24         if(StringUtils.isNullOrEmpty((content))
25
26             throw new Error('Content cannot be empty')
27
28         this.content = content;
29     }
30
31     getTitle(){
32         return this.title;
33     }
34
35     getContent(){
36         return this.content;
37     }
38 }
```

Prioritize composition over inheritance

Inheritance and composition are both very common techniques applied in code reuse. As we know, inheritance allows us to define an implementation from a parent class, while composition is based on assembling different objects to obtain more complex functionality.

Choosing composition over inheritance helps us keep each class encapsulated and focused on a single task (the principle of responsibility), favoring modularity and avoiding dependency coupling. A high amount of coupling not only forces dependencies on us that are not needed, but it also limits the flexibility of our code when introducing changes.

This is not to say that you should never use inheritance. There are situations in which inheritance works very well, the key is to know how to differentiate them. A good way to make this differentiation is by asking yourself whether the inheriting class really **is** a child or whether it just **has** elements from the parent. Let's see an example:

```
1  class Employee {
2      private name: string;
3      private email: string;
4
5      constructor(name:string, email:string) {
6          this.name = name;
7          this.email = email;
8      }
9
10     // ...
11 }
12
13 class EmployeeTaxData extends Employee {
14     private ssn: string;
15     private salary: number;
16
17     constructor(ssn:string, salary:number) {
18         super();
19         this.ssn = ssn;
20         this.salary = salary;
21     }
22     //...
23 }
```

As we can see, this is a somewhat forced example of misapplied inheritance, since in this case an employee "has" *EmployeeTaxData*, not "is" *EmployeeTaxData*. If we refactor by applying composition, the classes would be as follows:

```
1  class EmployeeTaxData{
2      private ssn: string;
3      private salary: number;
4
5      constructor(ssn:string, salary:number) {
6          this.ssn = ssn;
7          this.salary = salary;
8      }
9      //...
10 }
11
12 class Employee {
13     private name: string;
14     private email: string;
15     private taxData: EmployeeTaxData;
16
17     constructor(name:string, email:string) {
18         this.name = name;
19         this.email = email;
20     }
21
22     setTaxData(taxData:EmployeeTaxData){
23         this.taxData = taxData;
24     }
25     // ...
26 }
```

As we can see here, the responsibility of each of the classes is much more defined, and it also generates a code that is less coupled and modular.

Comments and format

Avoid using comments

“Don’t comment on badly written code, rewrite it” – Brian W. Kernighan

When you need to add comments to your code it is because this code is not self explanatory enough, which means that we are not choosing names that are good enough. When you see the need to write a comment, try to refactor your code and/or give the elements different names.

Often, when we use third-party libraries, APIs, frameworks, etc., we will find ourselves in situations where writing a comment will be better than leaving a complex solution or a hack without explanation. Ultimately, the idea is that comments are the exception, not the rule.

In any case, if you need to make use of comments, the important thing is to comment on the why, rather than comment on the what or the how, since the how we see, is the code, and the what should not be necessary if you write self-explanatory code. But the reason why you have decided to solve something in a certain way knowing that it is strange, should be explained.

Consistent format

“Good code always seems to be written by someone who cares” – Michael Feathers

In every software project there should be a series of simple guidelines that help us to harmonize the legibility of the code of our project, especially when we work as a team. Some of the rules that could be emphasized are:

Similar problems, symmetric solutions

It is essential to follow the same patterns when solving similar problems within the same project. For example, if we are solving a CRUD of one entity in a certain way, it is important that to implement the CRUD of other entities we keep to the same style.

Density, aperture and vertical distance

The lines of code with a direct relationship should be vertically dense, while the lines that separate concepts should be separated by white spaces. On the other hand, related concepts should be kept close to each other.

The most important comes first

The top elements of the files should contain the most important concepts and algorithms, and the details should be increased as we go down the file.

Indentation

Last but not least, we should respect indentation. We should indent our code according to its position, depending on whether it belongs to the class, to a function or to a block of code.

This is something that may seem like common sense, but I want to emphasize it because it is a common problem. I have an anecdote about that: when I was in the university I had a professor who, when you gave him an exercise with a bad indentation, would simply not correct it. After going through many projects I understood why.

Section II: SOLID Principles

Introduction to SOLID

In the section on Clean Code, we saw that the total cost of a software product is the sum of the development and maintenance costs, and that the maintenance cost is usually much higher than the initial development cost.

In that section we focused on the idea of minimizing the cost of maintenance corresponding to understanding the code, and now we are going to focus on how the SOLID principles can help us to write a more intuitive code that is testable and tolerant to changes.

Before we delve into SOLID, let's talk about what happens in our project when we write STUPID code.

From STUPID to SOLID

Don't be offended, I'm not being impolite, STUPID is simply an acronym based on six code smells that describe how the software we develop should NOT be.

- Singleton pattern
- Tight Coupling
- Untestability
- Premature optimization
- Indescriptive Naming
- Duplication

What is code smell?

The term “code smell”, or a bad smell in the code, was created by Kent Beck in one of the chapters of the famous book Refactoring by Martin Fowler. The concept, as you can imagine, is related to technical debt. In this case, the code smells refer to possible indications that something is not quite right in our code and that we will probably have to refactor it.

There are multiple code smells and we are going to focus on just a few of them. If you want to go deeper into the subject, I recommend that you read the “Bad Smells” chapter of the book [Refactoring](#)¹.

Singleton pattern

The singleton pattern is perhaps one of the best known and at the same time most reviled patterns. The intent of this pattern is to try to ensure that a class has a single instance and provide global access to it. It is usually implemented by creating a static variable in the class that stores an instance of itself. Said variable is initialized for the

first time in the constructor or in a named constructor.

<https://amzn.to/33GqLj9>

```
1 class Singleton {
2     constructor(){
3         if(Singleton.instance){
4             return Singleton.instance;
5         }
6
7         this.title = "my singleton";
8         Singleton.instance = this;
9     }
10 }
11
12 let mySingleton = new Singleton()
13 let mySingleton2 = new Singleton()
14
15 console.log("Singleton 1: ", mySingleton.title);
16 mySingleton.title = "modified in instance 1"
17 console.log("Singleton 2: ", mySingleton2.title);
```

You can access the interactive example from [here](#) ¹².

One of the reasons why using Singleton patterns is considered to be bad practice is because it generally exposes the instance of our object to the global context of the application, making it susceptible to modification at any time and loss of control.

Another reason is that doing unit testing can be hell because each test must be totally independent from the previous one and that can not be done, so by keeping the state, the application becomes difficult to test.

Although it can still be used, the separation of the lifecycle management of the class from the class itself is usually recommended.

Tight Coupling

Surely you have read or heard that a tight coupling between classes makes the maintainability and tolerance to change of a software project difficult, and that low

¹²<https://repl.it/@SoftwareCrafter/STUPID-Singleton>

coupling and good cohesion would be ideal. But what exactly do these concepts refer to?

Coupling and cohesion

Cohesion refers to the relationship between the modules of a system. In terms of class, we can say that it has high cohesion if its methods are closely related to each other. A code with high cohesion is usually more self-contained, that is, it contains all the pieces you need, therefore it is also usually easier to understand. However, if we increase cohesion too much, we tend to create modules with multiple responsibilities.

Coupling, on the other hand, refers to the relationship between the modules of a system and their dependence on each other. If we have many relationships between these modules, with many dependencies on each other, we will have a high degree of coupling. In contrast, if the modules are independent of each other, the coupling will be low. If we favor low coupling, we will get smaller modules with more defined responsibilities, but they would also be more dispersed.

The virtue is in the balance, that is why we should try to favor low coupling but without sacrificing cohesion.

Untestable code

Most of the time, code that is untestable or hard to test is caused by high coupling and / or when dependencies are not injected. We will deal more with this last concept when we talk about the SOLID principle of dependency inversion.

Although there are specific techniques to deal with these situations, it would be ideal if our design took them into account from the beginning in order to be able to carry out the tests. In this way we ensure that problematic situations such as high coupling or global state dependencies manifest themselves immediately. We will look at this in the Unit Testing and TDD section.

Premature optimization

“We will cross that bridge when we get to it”

Keeping options open by delaying decision making allows us to give more relevance to what is most important in an application: the business rules, that is where the value really lies. In addition, the simple fact of postponing these decisions will allow us to have more information about the real needs of the project, which will allow us to make better decisions since they will be based on any new requirements that might have arisen.

Donald Knuth used to say that premature optimization is the root of all evil. This does not mean that we should write poorly optimized software, but rather that we should not anticipate the requirements and develop unnecessary abstractions that may add accidental complexity.

Essential complexity and accidental complexity

The accidental design antipattern complexity refers to what occurs when a software product development is implemented with a solution of complexity greater than the minimum necessary.

Ideally, the complexity is inherent to the problem, this complexity is known as essential complexity. But, what usually happens is that we end up accidentally introducing complexity due to ignorance or team planning problems, which makes the project difficult to maintain and not very tolerant of change.

If you want to continue delving into these ideas, I recommend the article [No Silver Bullet - Essence and Accidents of Software Engineering](http://worrydream.com/refs/Brooks-NoSilverBullet.pdf)¹³. Its author and Alan Turing award winner, Fred Brooks, divided the properties of software into essential and accidental, based on Aristotle's decomposition of knowledge.

Indescriptive Naming

The next STUPID principle is Indescriptive Naming. It basically tells us that the names of variables, methods and classes must be selected with care so that they give expressiveness and meaning to our code. We have already dealt with this in the chapter on names and variables.

¹³<http://worrydream.com/refs/Brooks-NoSilverBullet.pdf>

Duplication

The last STUPID principle refers to the DRY principle (don't repeat yourself), which I already mentioned in the functions section. Basically, as a general rule, we should avoid duplicate code, although there are exceptions.

Real duplication

The code in an exact duplication, in addition to being identical, fulfills the same function. Therefore, if we make a change, we must propagate it manually to all parts of our project where the code is found, we must also change it in the same way, which increases the chances of human error occurring. This is the type of duplicate code that we should avoid and that we have to unify.

Accidental duplication

Unlike real duplication, accidental duplication is one in which the code may appear the same, but actually serves different functions. That is, in the case of there being a reason to change the code, it is likely that it will only be necessary to modify some of the places where said code is found.

SOLID principles to the rescue

The SOLID principles show us how to organize our functions and data structures into components and how these components should be interconnected. These components are usually classes, although this does not mean that these principles are only applicable to the object-oriented paradigm, since we could simply have a grouping of functions and data, for example, in a Closure. So, every software product has such components, whether they are classes or not, therefore it would make sense to apply the SOLID principles.

The acronym SOLID was created by Michael Feathers, and, of course, popularized by Robert C. Martin in his book *Agile Software Development: Principles, Patterns, and Practices*. It consists of five principles or conventions of software design, widely accepted by the industry, which have a goal of helping us to improve the maintenance costs derived from changing and testing our code.

- Single Responsibility
- Open/Closed
- Liskov substitution
- Interface segregation
- Dependency Inversion

It is important to emphasize that these are principles, not rules. A rule is mandatory, while principles are recommendations that can help make things better.

Single responsibility principle (SRP)

“There should never be more than one reason to change a class or module.” – Robert C. Martin

The first of the five principles, single responsibility principle (SRP), states that a class should only have one responsibility. At the end of the 80's, Kent Beck and Ward Cunningham already applied this principle through CRC (Class, Responsibility, Collaboration) cards, with which they detected responsibilities and collaborations between modules.

Having more than one responsibility in our classes or modules makes the code difficult to read, test and maintain. That is, it makes the code less flexible, more rigid, and ultimately less tolerant of change.

Most of the time, programmers misapply this principle, since we often confuse “having a single responsibility” with “doing one thing.” We have already seen a principle like this in the chapter on functions: functions should do one thing and do it well. We use this principle to refactor large functions into smaller ones, but this does not apply when designing classes or components

What do we understand by responsibility?

The single responsibility principle is not based on creating classes with a single method, but on designing components that are only exposed to one source of change. Therefore, the concept of responsibility refers to those actors (sources of change) that could request different modifications in a certain module depending on their role in the business. Let's see an example:

```
1  class UseCase{
2    doSomethingWithTaxes(){
3      console.log("Do something related with taxes ...")
4    }

5
6    saveChangesInDatabase(){
7      console.log("Saving in database ...")
8    }
9
10   sendEmail(){
11     console.log("Sending email ...")
12   }
13 }
14
15 function start(){
16   const myUseCase = new UseCase()
17
18   myUseCase.doSomethingWithTaxes();
19   myUseCase.saveInDatabase();
20   myUseCase.sendEmail();
21 }
22
23 start();
```

You can access the interactive example from [here](https://repl.it/@SoftwareCrafter/SOLID-SRP)¹⁴.

In this example we have a UseCase class that consists of three methods: doSomethingWithTaxes(), sendEmail() and saveChangesInDatabase(). At first glance, it can be seen that we are mixing three very different layers of architecture: business logic, presentation logic, and persistence logic. But, also, this class does not respect the principle of single responsibility because the operation of each of the methods is susceptible to being changed by three different actors.

The doSomethingWithTaxes() method could be specified by the accounting department, while sendEmail() could be subject to change by the marketing department

¹⁴<https://repl.it/@SoftwareCrafter/SOLID-SRP>

and, finally, the `saveChangesInDatabase()` method could be specified by the system's department.

These departments probably do not exist in your company, and the person in charge of assuming these roles for now might be you, but keep in mind that the needs of a project are evolving. It is for this reason that one of the most important values of the software is tolerance to change. Therefore, even if we are working on a small project, we must do the exercise of differentiating responsibilities to ensure that our software is flexible enough to meet any new needs that may appear.

Applying the SRP

Returning to the example, one way to separate these responsibilities could be by moving each of the functions of the `UseCase` class to others, like this :

```
1  class UseCase{
2      constructor(repo, notifier){
3          this.repo = repo;
4          this.notifier = notifier;
5      }
6
7      doSomethingWithTaxes(){
8          console.log("Do something related with taxes ...")
9      }
10
11     saveChanges(){
12         this.repo.update();
13     }
14
15     notify(){
16         this.notifier.notify("Hi!")
17     }
18 }
19
20 class Repository{
```

```
21     add(){
22         console.log("Adding in database");
23     }
24
25     update(){
26         console.log("Updating in database...");
27     }
28
29     remove(){
30         console.log("Deleting from database ...");
31     }
32
33     find(){
34         console.log("Finding from database ...");
35     }
36 }
37
38
39 class NotificationService{
40     notify(message){
41         console.log("Sending message ...");
42         console.log(message);
43     }
44 }
45
46
47 function start(){
48     const repo = new Repository()
49     const notifier = new NotificationService()
50     const myUseCase = new UseCase(repo, notifier)
51
52     myUseCase.doSomethingWithTaxes();
53     myUseCase.saveChanges();
54     myUseCase.notify();
55 }
56
```

```
57 start();
```

You can access the interactive example from [here](#)¹⁵.

The UseCase class would then represent a use case with a more defined responsibility, since now the only actor related to the classes is the one in charge of the specification of the `doSomethingWithTaxes()` operation. To do this we have extracted the implementation of the rest of the operations to the Repository and NotificationService classes.

The first one implements a repository (as you can see it is a fake repository) and is responsible for all operations related to persistence. On the other hand, the NotificationService class would deal with all the logic related to notifications to the user. In this way we would already have separated the three responsibilities that we had detected.

Both classes are injected via constructor to the UseCase class, but, as you can see, they are concrete implementations, so the coupling remains high. We will continue to examine this in the following chapters, especially in the one on dependency inversion.

Detecting violations of SRP

Knowing whether or not we are respecting the single responsibility principle can be somewhat ambiguous at times. Below we will see a list of situations that will help

us detect violations of the SRP:

- Too generic a name. Choosing an overly generic name often results in a God Object, an object that does too many things.
- Changes usually affect this class. When a high percentage of changes usually affect the same class, it may be a sign that the class is too coupled or has too many responsibilities.
- The class involves multiple layers of architecture. If, as we saw in the case of the example, our class does things like access the persistence layer or notify the user, in addition to implementing the business logic, it is clearly violating the

¹⁵<https://repl.it/@SoftwareCrafter/SOLID-SRP2>

- High number of imports. Although this by itself does not imply anything, it could be a symptom of violation.
- High number of public methods. When a class has an API with a high number of public methods, it is often a sign that it has too many responsibilities.
- Excessive number of lines of code. If our class only has a single responsibility, its number of lines should not, in principle, be very high.

OCP - Open/Closed principle

“All software entities should be open for extension, but closed for modification”.
– Bertrand Meyer

The Open-Closed principle, set forth by Bertrand Meyer, recommends that, in cases where new behaviors are introduced into existing systems, instead of modifying old components, new components should be created. The reason is that if those components or classes are being used in another part (of the same project or of others) we will be altering their behavior and thus could cause unwanted effects.

This principle guarantees improvements in the stability of your application as it tries to prevent existing classes from changing frequently, which also makes dependency chains a bit less fragile as there will be fewer moving parts to worry about. When we create new classes it is important to take this principle into account in order to facilitate their extension in the future. But, in practice, how is it possible to modify the behavior of a component or module without modifying the existing code?

Applying OCP

Although this principle may seem like a contradiction in itself, there are several ways of applying it, but all of them depend on our context. One of these techniques could be to use extension mechanisms, such as inheritance or composition, to use those classes while we modify their behavior. As we discussed in the classes chapter in the Clean Code section, you should try to prioritize composition over inheritance.

I think a good context to illustrate how to apply the OCP might be to try to decouple an infrastructure element from the domain layer. Imagine that we have a task management system, specifically a class called *TodoService*, which is responsible for making an HTTP request to a REST API to obtain the different tasks that the system contains:

```
1  const axios = require('axios');
2
3  class TodoExternalService{
4
5      requestTodoItems(callback){
6          const url = 'https://jsonplaceholder.typicode.com/todos/';
7
8          axios
9              .get(url)
10             .then(callback)
11         }
12     }
13
14     new TodoExternalService()
15         .requestTodoItems(response => console.log(response.data))
```

You can access the interactive example from [here](#)¹⁶.

In this example two things are happening, on the one hand we are coupling an infrastructure element and a third-party library in our domain service and, on the other, we are skipping the open / closed principle, since if we wanted to replace the library *axios* for another, like *fetch*, we would have to modify the class. To solve these problems we are going to use the adapter pattern.

Adapter pattern

The adapter pattern belongs to the category of structural patterns. It is a pattern in charge of homogenizing APIs, this facilitates the task of decoupling both elements from different layers of our system and third-party libraries.

To apply the adapter pattern in our example, we need to create a new class that we are going to call *ClientWrapper*. This class will expose a *makeRequest* method that will be in charge of making requests for a specific URL received as a parameter. You will also receive a callback in which the request will be resolved:

¹⁶<https://repl.it/@SoftwareCrafter/SOLID-OCP-2>


```
1 class ClientWrapper{
2   makeGetRequest(url, callback){
3     return axios
4       .get(url)
5
6   } .then(callback);
7 }
```

ClientWrapper is a class that belongs to the infrastructure layer. To use it in our domain in a decoupled way, we must inject it via the constructor (we will examine the injection of dependencies in the chapter about dependency inversion principle). It is this easy:

```
1 //infrastructure/ClientWrapper
2
3 const axios = require('axios');
4
5 export class ClientWrapper{
6   makeGetRequest(url, callback){
7     return axios
8       .get(url)
9       .then(callback);
10  }
11
12 //domain/ToDoService
13 export class ToDoService{
14   client;
15
16   constructor(client){
17     this.client = client;
18   }
19
20   requestTodoItems(callback){
21     const url = 'https://jsonplaceholder.typicode.com/todos/';
22     this.client.makeGetRequest(url, callback)
23   }
```

```
24 }
25
26 //index
27 import {ClientWrapper} from '../infrastructure/ClientWrapper'
28 import {TodoService} from '../domain/TodoService'
29
30 const start = () => {
31   const client = new ClientWrapper();
32   const todoService = new TodoService(client);
33
34   todoService.requestTodoItems(response => console.log(response.data))
35 }
36
37 start();
```

You can access an interactive example [from here](#)¹⁷

As you can see, we have removed the *axios* dependency from our domain. Now we could use our *ClientWrapper* class to make HTTP requests throughout the project. This would allow us to maintain a low coupling with third-party libraries, which is tremendously positive for us, since if we wanted to change the *axios* library to *fetch*, for example, we would only have to do it in our *ClientWrapper* class:

```
1 export class ClientWrapper{
2   makeGetRequest(url, callback){
3     return fetch(url)
4       .then(response => response.json())
5       .then(callback)
6   }
7 }
```

In this way we have managed to change *requestTodoItems* without modifying its code, thus respecting the open/closed principle.

¹⁷<https://repl.it/@SoftwareCrafter/SOLID-OCP1>

Detect violations of OCP

As you may have seen, this principle is closely related to that of single responsibility. Normally, if a high percentage of changes is affecting our class, it is a symptom that this class, in addition to being too coupled and having too many responsibilities, is violating the open closed principle.

Also, as we saw in the example, the principle is violated very often when we involve different layers of the project architecture.

LSP - Liskov Substitution Principle

“Functions that use pointers, or references to base classes, should be able to use derived class objects without knowing it.” - Robert C. Martin

The third SOLID principle takes its name from Dr. Barbara Jane Huberman, better known as Barbara Liskov. This renowned American software engineer, in addition to being the winner of a Turing Prize (the Nobel Prize for Computer Science), she was the first woman in the United States to receive a Ph.D. in Computer Science.

Barbara Liskov and Jeanette Wing, together, defined this principle in 1994, which goes something like this: if U is a subtype of T , any instance of T should be able to be substituted by any instance of U without altering the properties of the system. In other words, if a class A is extended by a class B , we should be able to replace any instance of A with any object of B without the system crashing or unexpected behaviors happening.

This principle refutes the preconceived idea that classes are a direct way of modeling the world, which couldn't be further from reality. Now we will see why, with the typical example of the rectangle and the square.

Applying LSP

A square, from a mathematical point of view, is exactly the same as a rectangle, since a square is a rectangle with all sides equal. Therefore, we could model a square by extending a rectangle class, so that:

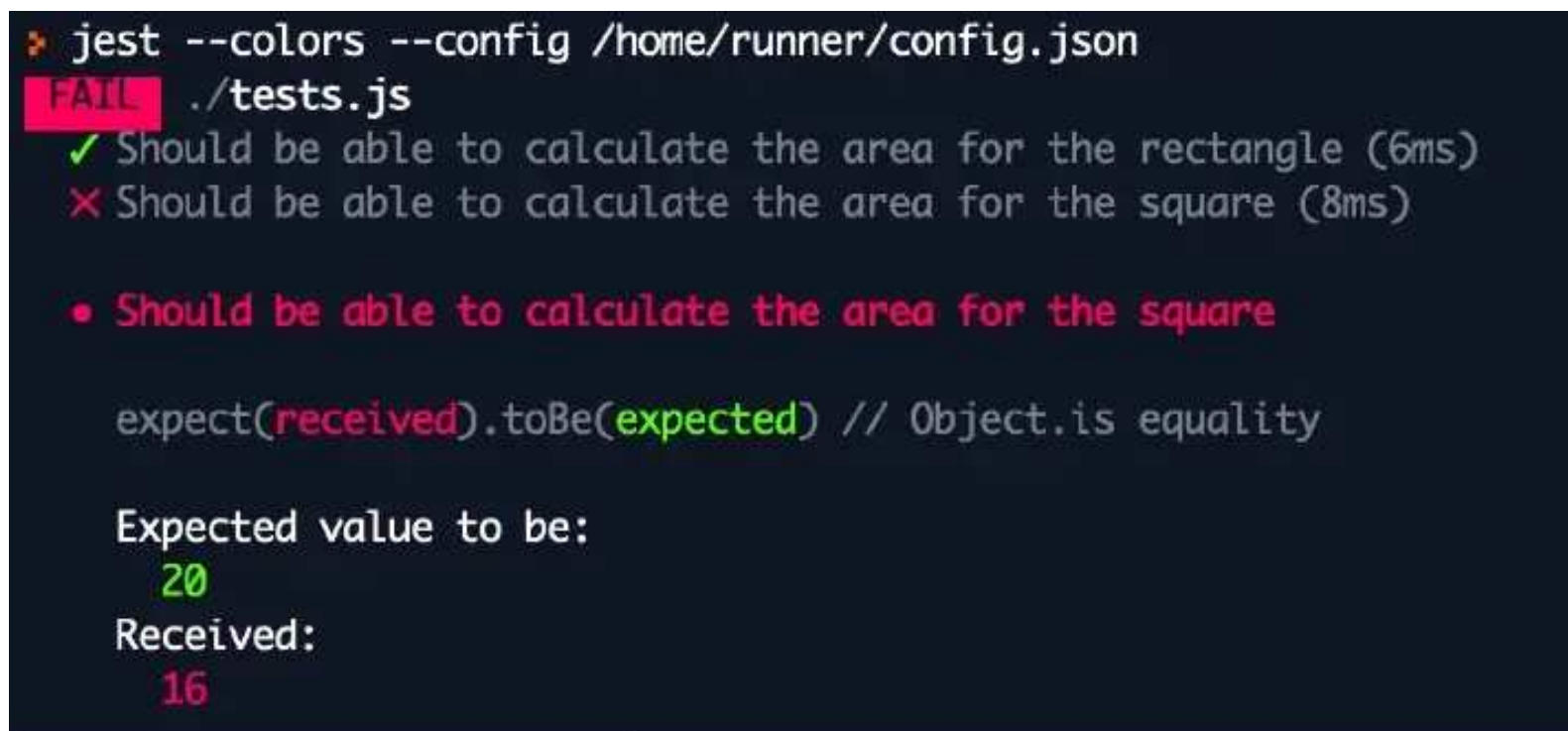
```
1  class Rectangle {
2      constructor() {
3          this.width = 0;
4          this.height = 0;
5      }
6
7      setWidth(width) {
8          this.width = width;
9      }
10
11     setHeight(height) {
12         this.height = height;
13     }
14
15     getArea() {
16         return this.width * this.height;
17     }
18 }
19
20
21 class Square extends Rectangle {
22     setWidth(width) {
23         this.width = width;
24         this.height = width;
25     }
26
27     setHeight(height) {
28         this.width = height;
29         this.height = height;
30     }
31 }
```

In the case of the square, the width is the same as the height, so each time we call *setWidth* or *setHeight* we set the same value for width and height. A priori, this might seem like a valid solution. We are going to create a unit test (we will study unit tests in the section dedicated to *testing*) to verify that the *getArea()* method returns the

correct result:

```
1 test('Should be able to calculate the area for the rectangle', ()=>{
2   let rectangle = new Rectangle()
3   rectangle.setHeight(5)
4   rectangle.setWidth(4)
5
6   expect(rectangle.getArea()).toBe(20)
7 })
```

When we run the test, it will pass successfully. But what would happen if we replace the class *Rectangle* with *Square*? Well, the test will fail, since the expected result would be 16 instead of 20. We would, therefore, be violating the Liskov substitution principle. You can test the example code from [here](#)¹⁸.

A screenshot of a terminal window showing Jest test results. The command 'jest --colors --config /home/runner/config.json' is run. The output shows a 'FAIL' status for './tests.js'. Two test cases are listed: 'Should be able to calculate the area for the rectangle (6ms)' which passes with a green checkmark, and 'Should be able to calculate the area for the square (8ms)' which fails with a red X. A detailed error message for the failing test is shown: 'Should be able to calculate the area for the square'. It indicates that the expected value is 20 (in green) and the received value is 16 (in red). The error message also includes 'expect(received).toBe(expected) // Object.is equality'.

Example result

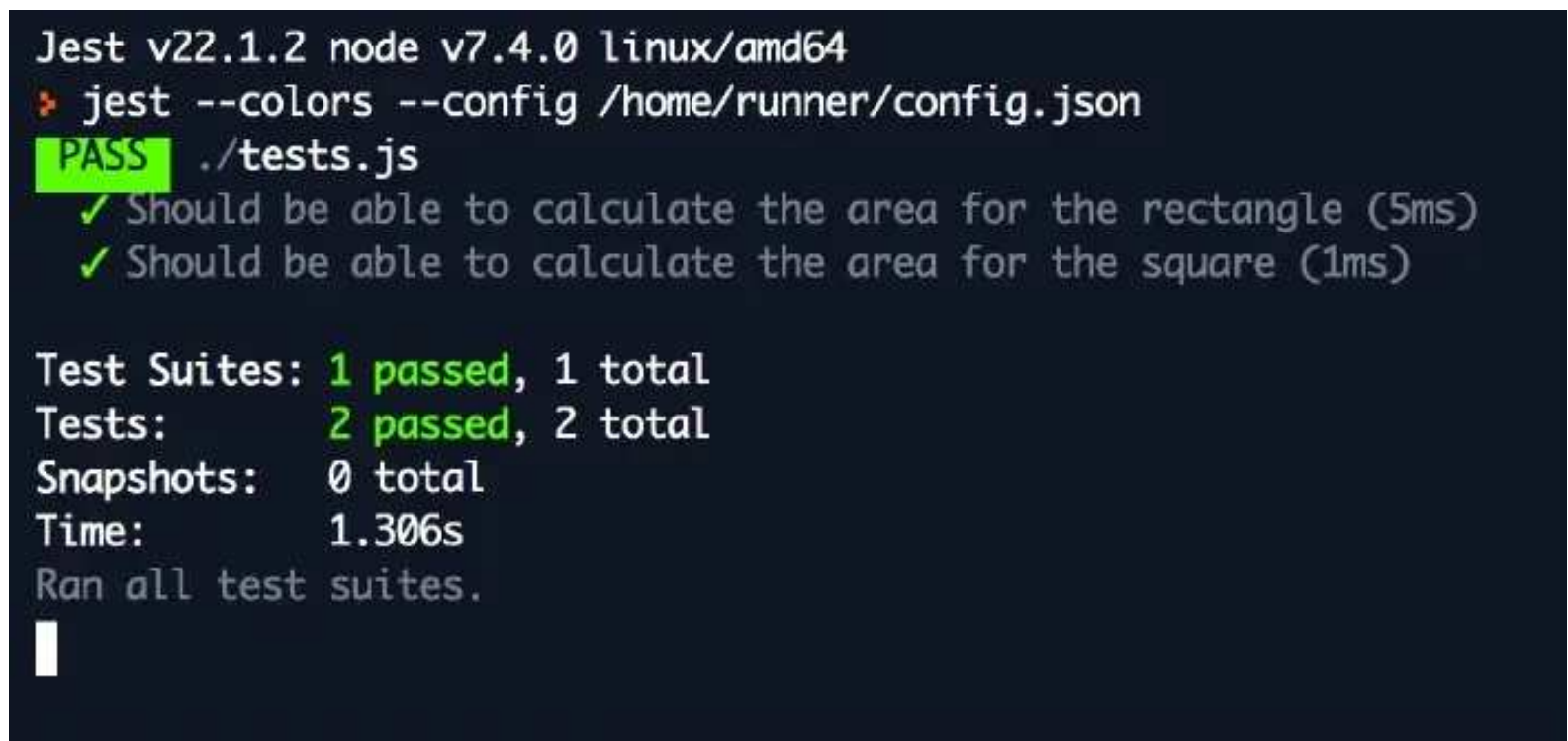
As we can see, the problem is that we are forced to reimplement the public *setHeight* and *setWidth* methods. These methods make sense in the *Rectangle* class, but they don't make sense in the *Square* class. A possible solution for this could be to create a different class hierarchy, extracting a parent class that has common features and modeling each child class according to its specifications:

¹⁸<https://repl.it/@SoftwareCrafter/SOLID-LSP>

```
1  class Figure{
2      constructor() {
3          this.width = 0;
4          this.height = 0;
5      }
6
7      getArea() {
8          return this.width * this.height;
9      }
10 }
11
12
13 class Rectangle extends Figure {
14     constructor(width, height) {
15         super();
16         this.width = width;
17         this.height = height;
18     }
19 }
20
21
22 class Square extends Rectangle {
23     constructor(length) {
24         super();
25         this.width = length;
26         this.height = length;
27     }
28 }
29
30
31 test('Should be able to calculate the area for the rectangle', ()=>{
32     let rectangle = new Rectangle(5, 4)
33
34     expect(rectangle.getArea()).toBe(20)
35 })
36
```

```
37 test('Should be able to calculate the area for the square', ()=>{
38   let square = new Square(5)
39
40   expect(square.getArea()).toBe(25)
41 })
```

We have created a *Figure* class from which the *Square* and *Rectangle* classes inherit. In these child classes, the methods to set the width and height are no longer exposed, so they are perfectly interchangeable with each other, therefore it complies with the principle of LSP. You can access the interactive example from [here](#)¹⁹.

A terminal window with a dark background showing the output of a Jest command. The command is 'jest --colors --config /home/runner/config.json ./tests.js'. The output shows 'PASS' in green, followed by two green checkmarks indicating successful tests: 'Should be able to calculate the area for the rectangle (5ms)' and 'Should be able to calculate the area for the square (1ms)'. Below this, a summary shows 'Test Suites: 1 passed, 1 total', 'Tests: 2 passed, 2 total', 'Snapshots: 0 total', and 'Time: 1.306s'. The final line is 'Ran all test suites.'.

```
Jest v22.1.2 node v7.4.0 linux/amd64
> jest --colors --config /home/runner/config.json
PASS ./tests.js
  ✓ Should be able to calculate the area for the rectangle (5ms)
  ✓ Should be able to calculate the area for the square (1ms)

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        1.306s
Ran all test suites.
```

Example result

Anyway, if you think about it, this is a forced case of inheritance because the area calculation method of the *Figure* class would only work with squares and rectangles. In this case, a better solution would be to use the TypeScript interfaces to define a contract and apply polymorphism. We will see this in the chapter on the principle of interface segregation.

¹⁹<https://repl.it/@SoftwareCrafter/SOLID-LSP-2>

Detect violations of LSP

As we have just seen, the easiest way to detect violations of the Liskov substitution principle is by observing if the overridden methods in a child class behave as expected. A very common form of LSP violation is when overridden methods of a child class return *null* or throw an exception.

ISP - Interface segregation principle

“Clients shouldn’t be forced to depend on interfaces they don’t use.” - Robert C. Martin

The principle of interface segregation was defined by Robert C. Martin when he worked at Xerox as a consultant. This principle indicates that a class should not depend on methods or properties that it does not need. Therefore, when we define the contract of an interface, we must focus on the classes that are going to use it (the interfaces belong to the client class), not on the implementations that are already developed.

In languages that do not have interfaces, such as JavaScript, this principle does not make much sense and the good work of the developer himself is often relied upon to apply the concept of *duck typing* consistently. This means that the methods and properties of an object determine its semantic validity, rather than its class hierarchy or the implementation of a specific interface.

In the examples in this chapter we are going to use TypeScript and its interfaces. Not only do these help us to better understand this principle, but they are also a very powerful tool when defining contracts in our code.

Applying the ISP

Interfaces are abstractions that define the behavior of the classes that implement them. The problem appears when these interfaces try to define more methods than necessary, since the classes that implement them will not need these methods and we will be forced to create forced implementations for them, and they will often throw an exception, which will end up violating the principle of Liskov substitution.

Let's see this with an example: imagine that we need to design a system that allows us to control a car in a basic way, regardless of the model, so we define an interface like this:

```
1 interface Car{
2   accelerate: () => void;
3   brake:() => void;
4   startEngine: () => void;
5 }
```

Now we will define a Mustang class that implements this interface:

```
1 class Mustang implements Car{
2   accelerate(){
3     console.log("Speeding up...")
4   }
5
6   brake(){
7     console.log("Stopping...")
8   }
9
10  startEngine(){
11    console.log("Starting engine... ")
12  }
13 }
```

So far so good. But suddenly one day our system reaches the ears of Elon Musk and he wants us to adapt it to his company, Tesla Motors. As you know, Tesla, in addition to the electrical component of its vehicles, has some differentiating features from other car companies, such as the *auto pilot* and the Ludicrous Speed mode. So, of course we adapted our system to control, in addition to the current vehicles, those of our friend Elon Musk.

To do this, we add the new behavior associated with the new client to the *Car* interface:

```
1 interface Car{
2   accelerate: () => void;
3   brake:() => void;
4   startEngine: () => void;
5   autoPilot: () => void;
6   ludicrousSpeed: () => void;
7 }
```

We implement the modified interface in a new *ModelS* class:

```
1 class ModelS implements Car{
2   accelerate(){
3     console.log("Speeding up...")
4   }
5
6   brake(){
7     console.log("Stopping...")
8   }
9
10  startEngine(){
11    console.log("Starting engine... ")
12  }
13
14  ludicrousSpeed(){
15    console.log("woooooooooow ...")
16  }
17
18  autoPilot(){
19    console.log("self driving... ")
20  }
21 }
```

But what about the *Mustang* class now? Well, the TypeScript compiler forces us to implement additional methods to fulfill the contract that we have defined in the Car interface:

```
1  class Mustang implements Car{
2      accelerate(){
3          console.log("Speeding up...")
4      }

5
6      brake(){
7          console.log("Stopping...")
8      }
9
10     startEngine(){
11         console.log("Starting engine... ")
12     }
13
14     ludicrousSpeed(){
15         throw new Error("Unsupported operation")
16     }
17
18     autoPilot(){
19         throw new Error("Unsupported operation")
20     }
21 }
```

Now we comply with the interface, but to do this we have had to implement the `autoPilot()` and `ludicrousSpeed()` methods forcibly. When we do this we are clearly violating the interface segregation principle, since we are forcing the Client class to implement methods that it cannot use.

The solution is simple, we can divide the interface into two pieces, one for the basic behaviors of any vehicle (Car) and another more specific interface (Tesla) that describes the behavior of the brand's models.

You can access the interactive editor with the complete example from [here](https://repl.it/@SoftwareCrafter/SOLID-ISP3)²⁰.

²⁰<https://repl.it/@SoftwareCrafter/SOLID-ISP3>

```
1 interface Car{
2     accelerate: () => void;
3     brake:() => void;
4     startEngine: () => void;
5 }
6
7 interface Tesla{
8     autoPilot: () => void;
9     ludicrousSpeed: () => void;
10 }
```

Finally, we must first refactor the Mustang class, so that it only implements Car and then the ModelS class, so that it implements both the Car and Tesla interfaces.

```
1 class Mustang implements Car{
2     accelerate(){
3         console.log("Speeding up...")
4     }
5
6     brake(){
7         console.log("Stopping...")
8     }
9
10    startEngine(){
11
12    } console.log("Starting engine... ")
13 }
14
15 class ModelS implements Car, Tesla{
16     accelerate(){
17         console.log("Speeding up...")
18     }
19
20     brake(){
21         console.log("Stopping...")
22     }
```

```
23
24     startEngine(){
25         console.log("Starting engine... ")
26     }

27
28     ludicrousSpeed(){
29         console.log("woooooooooow ...")
30     }
31
32     autoPilot(){
33         console.log("self driving... ")
34     }
35 }
```

It is important to be aware that dividing the interface does not mean that we divide its implementation. When applying the idea that a single class implements multiple specific interfaces, the interfaces are often referred to as a role interface.

Detect violation of ISP

As you can guess, this principle is closely related to that of single responsibility and Liskov substitution principles. Therefore, if the interfaces we design force us to violate these principles, it is very likely that you are also violating the ISP. If you

keep your interfaces simple and specific and, especially, keep in mind which client class is going to implement them, it will help you to respect this principle.

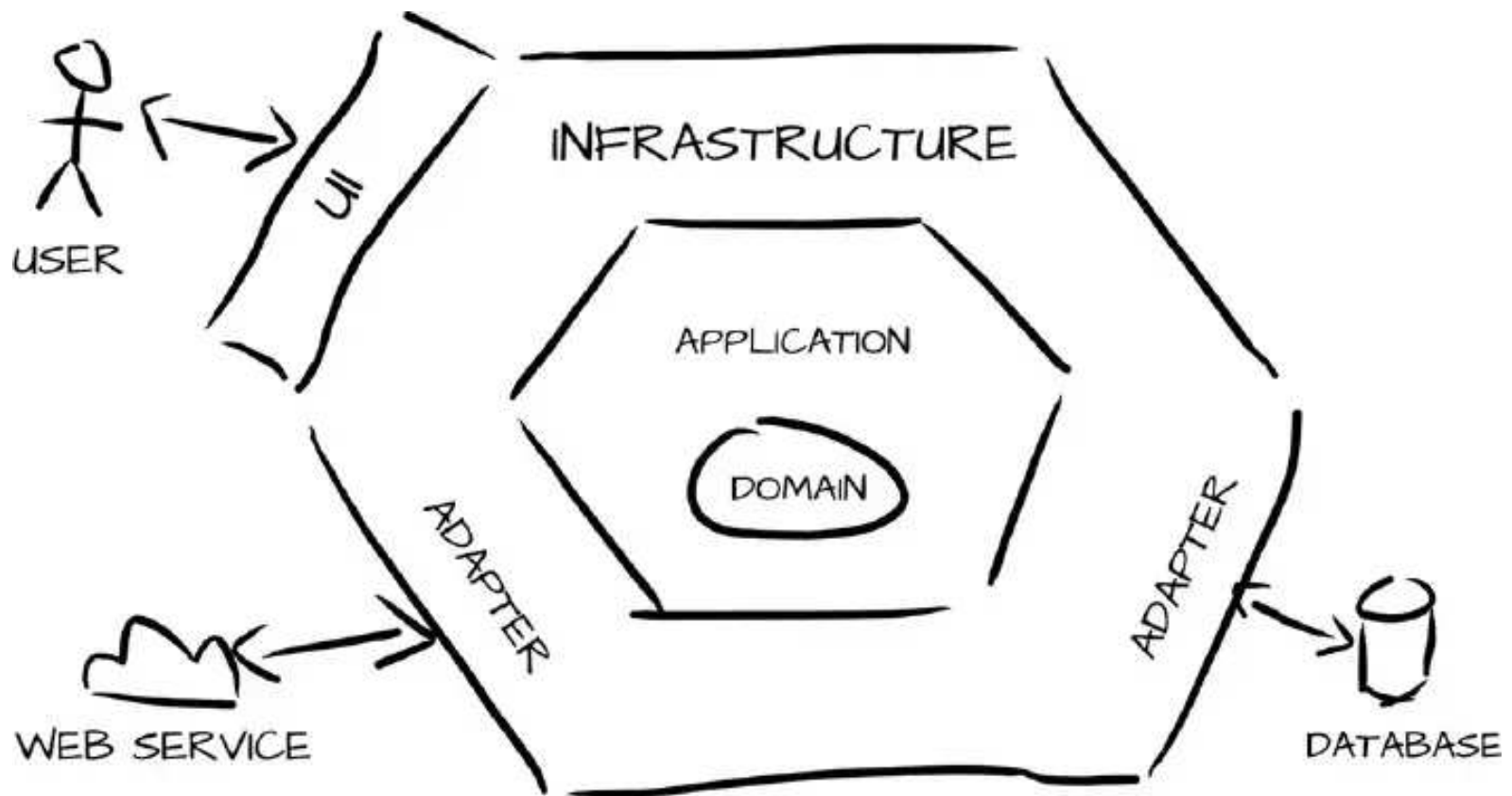
DIP - Dependency Inversion Principle

“High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.” - Robert C. Martin

In this chapter we are going to discuss the fifth and final principle, dependency inversion. This principle was created by Robert C. Martin in 1995, and it states that the classes or modules in the high-level layers should not depend on the classes or modules in the low-level layers, but both should depend on abstractions. Also, these abstractions should not depend on the details, but it is the details that should depend on them. But what does this mean? What does Robert C. Martin mean by low-level and high-level modules? What's the point of depending on abstractions?

High-level modules and low-level modules

When Uncle Bob says that high-level modules should not depend on low-level modules, he means that important components (high-level layers) should not depend on less important components (low-level layers). From the point of view of the hexagonal architecture, the most important components are those focused on solving the problem underlying the business, that is, the domain layer. The least important are those that are close to the infrastructure, that is, those related to UI, persistence, communication with external APIs, etc. But why is this so? Why is the infrastructure layer less important than the domain layer?



Hexagonal Architecture Diagram

Imagine that in our application we use a file persistence system, but for performance or scalability reasons we want to use a document database such as MongoDB. If we have correctly decoupled the persistence layer, for example applying the repository pattern, the implementation of this layer should be indifferent to the business rules (domain layer). So switching from one persistence system to another, once the repository is implemented, becomes practically trivial. On the other hand, a modification of the business rules could affect what data should be stored, thus affecting the persistence layer.

Exactly the same happens with the presentation layer; our domain layer should not care if we use *React*, *Vue* or *Angular*. Whatismore, although not a realistic scenario, we should even be able to replace the library that we use in our views, for example *React*, with *Vue* or *Angular*. On the other hand, a modification in the business rules would probably be reflected in the UI.

Depending on Abstractions

When we talk about abstractions we are referring to abstract classes or interfaces. One of the most important reasons why business rules or domain layers should depend on these and not on specific implementations is that it increases the tolerance for change. But why do we get this benefit?

Each change in an abstract component implies a change in its implementation. In contrast, changes to specific implementations, most of the time, do not require changes to the interfaces you implement. Therefore, abstractions tend to be more stable than implementations. Thus, if our domain depends on interfaces, it will be more tolerant to change, as long as they are designed respecting the Liskov substitution principle and that of interface segregation.

But how do we write our code to depend on abstractions and not on specific implementations? Don't be impatient, we still have to introduce one more concept, dependency injection.

Dependency Injection

In programming we refer to **dependency** when a module or component requires another in order to carry out its work. We say that a component *A* has a dependency on another component *B*, when *A* uses *B* to perform some task. This dependency manifests itself because component *A* cannot function without component *B*.

Dependencies in the *software* are necessary. The problem with these comes from the degree of coupling that the dependency has with the component. As we saw in the chapter on dependency and cohesion, we should try to favor a low degree of coupling, but without sacrificing cohesion. Let's analyze the following example:

```
1 class UseCase{
2     constructor(){
3         this.externalService = new ExternalService();
4     }
5
6     doSomething(){
7         this.externalService.doExternalTask();
8     }
9 }
10
11 class ExternalService{
12     doExternalTask(){
13         console.log("Doing task...")
14     }
15 }
```

In this case we are faced with a situation of high coupling, since the *UseCase* class has a **hidden dependency** on the *ExternalService* class. If for some reason we were to change the implementation of the *ExternalService* class, the functionality of the *UseCase* class could be affected. Could you imagine the nightmare that this would entail at the maintenance level in a real project? To deal with this problem we must start by applying the **dependency injection pattern**.

The term was coined by Martin Fowler. This is a design pattern that takes responsibility for instantiating one component to delegate it to another. Although it may sound complex it is very simple, let's see how to apply it in the example:

Coupled code with visible dependency

```
1 class UseCase{
2     constructor(externalService: ExternalService){
3         this.externalService = externalService;
4     }
5
6     doSomething(){
7         this.externalService.doExternalTask();
8     }
9 }
```

```
10
11 class ExternalService{
12     doExternalTask(){
13         console.log("Doing task...")
14     }
15 }
```

That's it, it's that simple, dependency injection via constructor , **which could also be done via the ** setter method**. Now, **although we still have a high degree of coupling, the **dependency is visible**, which makes the relationship between the classes clearer.



UML Diagram about visible dependencies

As you can see, it is a very simple concept but one that authors often complicate in the explanation.

Applying DIP

In our example we still have a high degree of coupling, since the *UseCase* class makes use of a specific implementation of *ExternalService* . *The ideal here is that the client class* (*UseCase*) depends on an abstraction (interface) that defines the contract it needs, in this case *doExternalTask()*, that is, the less important *ExternalService class*, must adapt to the needs of the most important class, *UseCase*.

```
1  interface IExternalService{
2      doExternalTask: () => void;
3  }
4
5  class UseCase{
6      externalService: IExternalService;
7
8      constructor(externalService: IExternalService){
9          this.externalService = externalService;
10     }
11
12     doSomething(){
13         this.externalService.doExternalTask();
14     }
15 }
16
17 class ExternalService implements IExternalService {
18     doExternalTask(){
19         console.log("Doing external task...")
20     }
21 }
22
23 const client = new UseCase(new ExternalService());
24
25 client.doSomething();
```

You can access the interactive example from [here](https://repl.it/@SoftwareCrafter/SOLID-DIP)²¹.

Now the code of the *UseCase* class is totally decoupled from the *ExternalService* class and only depends on an interface that was created based on your needs, and this means we can say that we have **inverted the dependency**.

²¹<https://repl.it/@SoftwareCrafter/SOLID-DIP>



UML diagram of inverted dependencies

Detecting violations of DIP

Perhaps the best piece of advice to help detect if we are violating the dependency inversion principle is to check that, in the architecture of our project, the high-level elements do not have dependencies on the low-level elements. From the point of view of the hexagonal architecture, this refers to the fact that the domain layer must not know about the existence of the application layer, and, in the same way, the elements of the application layer must not know anything about the infrastructure layer. In addition, dependencies with third-party libraries must be in the latter.

Section III: Testing & TDD

Introduction to testing

“Software testing can verify the presence of errors but not the absence of them”.
- Edsger Dijkstra

The first of Kent Beck’s four simple design rules tells us that our code must pass the set of automatic tests correctly. For Kent Beck this is the most important rule and it makes perfect sense, since creating a great design at the architectural level, or applying all the good practices that we have seen until now, is of no use if you can’t verify that your system works.



Cartoon about the importance of tests.

Software testing is very important when we work with dynamic languages like JavaScript, especially when the application acquires a certain complexity. The main reason for this is that there is no compilation phase like there is in static typing languages, so we cannot detect bugs until the moment we run the application.

This is one of the reasons why the use of TypeScript is so interesting, since the first error control is carried out by its compiler. This is not to say that we don't have testing if we use it, but rather that in my opinion, the ideal is to combine their use to get the best of both worlds.

Now we will see some general concepts about *testing of software*, such as the different types of tests available. Then we will focus on the ones that are most important from the developer's point of view: unit tests.

Types of software tests

Although in this section we are going to focus on the tests that developers write, specifically on unit tests, I think it is interesting to make a general classification of the different types that exist. To do this, the first thing to do is to answer this question:

What do we understand by testing?

The *testing of software* is a set of techniques used to verify that the system developed, either by us or by third parties, complies with the established requirements.

Manual tests vs automatic tests

The first great differentiation that we can make is taking into account how its execution is carried out, that is, whether it is done manually or automatically. Manual testing consists of preparing a series of cases and executing the necessary elements by hand, and as you can imagine it has many limitations: they are slow, difficult to replicate, expensive and also cover very few cases. That is why most tests should be automatic.

However we must be aware that not everything is automatable, since sometimes there are cases in which it becomes very expensive, or even directly impossible, to automate certain situations, so there is no choice but to do certain tests manually. This explains the importance of having a good QA team that complements the development team.

Functional vs non-functional tests

Perhaps a more intuitive way to classify the different existing types of software tests is by grouping them into functional tests and non-functional tests.

Functional tests

Functional tests refer to tests that verify the correct behavior of the system, subsystem or * software * component. That is, they validate that the code complies with the specifications that come from the business, and also that it is free of bugs. Within these types of test we mainly find the following:

- **Unit Tests:** These types of tests check basic elements of our software in isolation. They are the most important tests when it comes to validating the business rules that we have developed. We will focus on this type of testing throughout the testing section.
- **Integration tests:** Integration tests are those that test sets of basic elements, some infrastructure elements are usually included in this type of test, such as databases or API calls.
- **System tests:** These types of tests, also called end-to-end, test multiple elements of our architecture by simulating the behavior of an actor with our software.
- **Regression tests:** These types of test verify the functionality already delivered, that is, they are tests that are used to detect that the changes introduced in the system do not generate unexpected behavior. In short, any type of functional test that we have seen could be a regression test, as long as they have been passed correctly at some point and, after making some changes to the system, they begin to fail.

In addition to these types of functional tests you can find some more with different nomenclature such as: sanity testing, smoke testing, UI testing, Beta/Acceptance testing, etc. All of them belong to one or more of the previous types of functional tests.

Non-functional tests

The objective of non-functional tests is the verification of a requirement that specifies criteria that can be used to judge the operation of a system, such as availability, accessibility, usability, maintainability, security and / or performance. That is, unlike the functional tests, they focus on checking how the system responds, not on what it does or should do.

We can classify non-functional tests according to the type of non-functional requirement they cover, among all of these the following stand out:

- **Load tests:** They are tests by which the behavior of a software system is observed under different numbers of requests during a certain time.
- **Speed tests:** They check if the system generates the results in an acceptable time.
- **Usability tests:** These are tests that try to evaluate the UX of the system.
- **Security tests:** This is a set of tests which try to evaluate whether the developed system is exposed to known vulnerabilities.

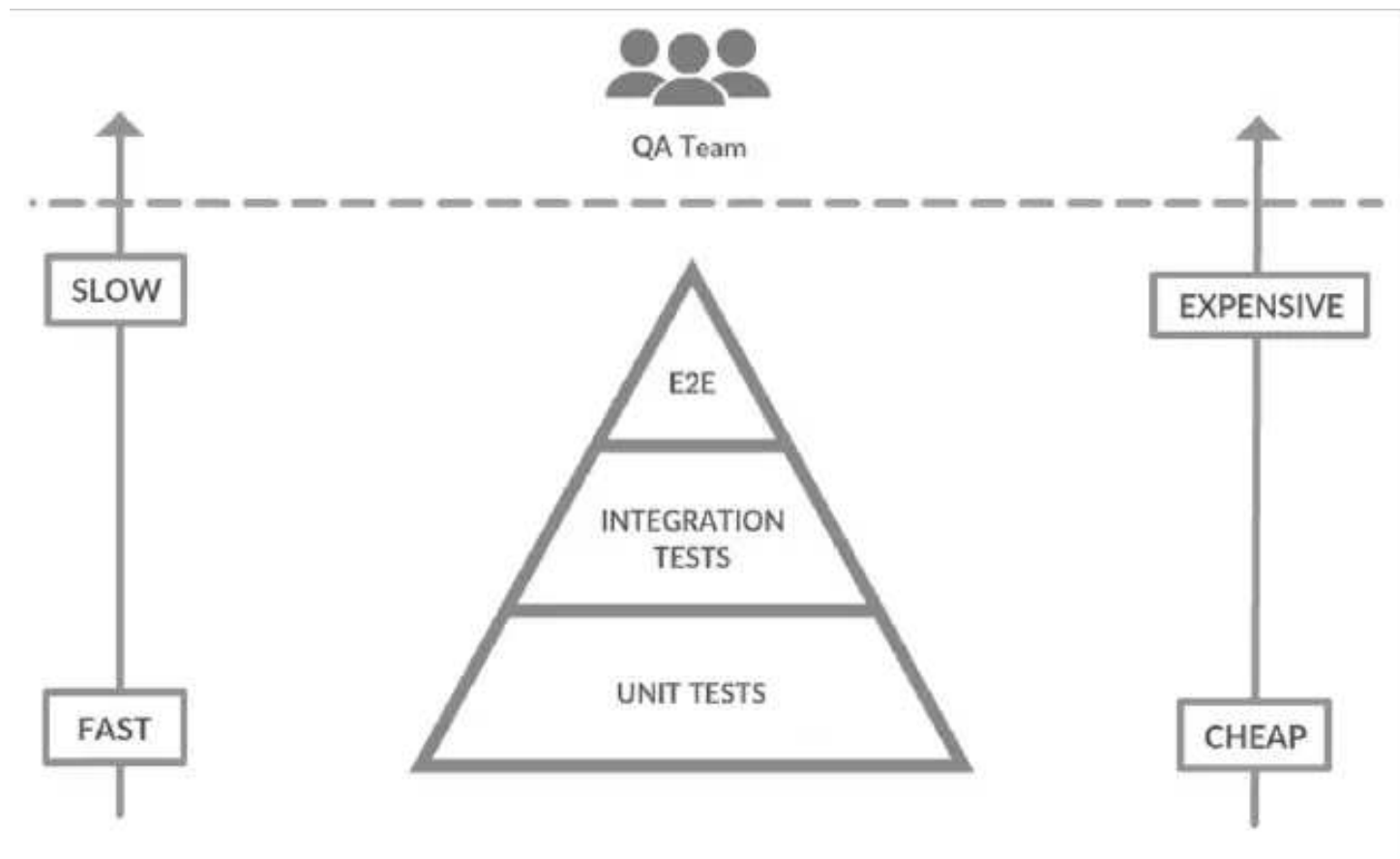
Both non-functional and functional tests follow the same process when generating them:

- **Initial scenario:** A series of input data is created to be able to run the tests.
- **Test execution:** The test with its corresponding input data is executed on the system.
- **Evaluation of the result:** The result obtained is analyzed to see if it matches what is expected.

Normally non-functional tests are generated by the QA team, while functional tests are usually created by developers, especially unit tests, integration tests and most system tests.

The Pyramid of testing

The Testing Pyramid, also known as Cohn's Pyramid (after its author Mike Cohn), is a very extended and accepted way of organizing functional tests at different levels, following a pyramid-shaped structure:



Pyramid of Testing.

The pyramid is very simple to understand, the idea is to try to organize the amount of tests we have, based on their execution speed and the cost of creating and maintaining them. That is why unit tests appear at the base of the pyramid, since, if we design them focusing on a single software unit in isolation, they are very fast to execute, easy to write and cheap to maintain.

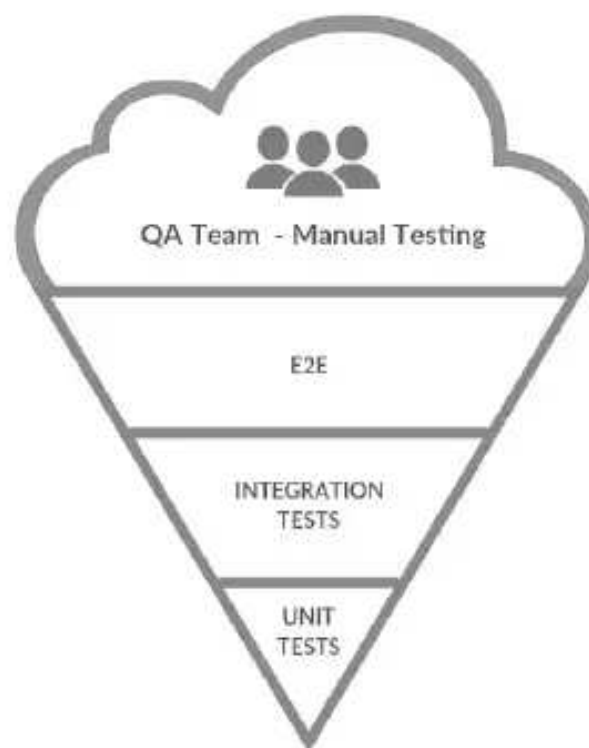
At the other extreme are the end-to-end or system tests. As we have mentioned, in these tests our system is tested from end to end, so all the elements of the system involved in a specific action come into play, therefore these tests seem slow to execute and complicated to create and maintain. We generally try not to have many of these tests due to their fragility and high maintenance cost.

The middle part of the pyramid is made up of integration tests, the objective of this type of test is to check if the different software units interact with certain infrastructure elements such as external APIs or databases in the expected way. These types of test are slower to execute and more complex to write and maintain than unit tests, although in many contexts they do also provide greater security. On the other hand, this type of test is much cheaper and faster than those of systems, for this reason it is ideal to have an intermediate amount of these.

Ice Cream cone anti-pattern

The Testing Pyramid should be understood as a recommendation and it does not have to always fit with our context, especially with regard to the number of unit and integration tests. For example, it might be the case that the software system we are designing doesn't have many business rules, and they can be covered by just a few unit tests. On the other hand, said system could be very demanding at the level of external elements, in which case it would probably be in our interest to have more integration tests than unit ones.

Usually, contexts such as the above are the exception and, even today, in many projects we find the dreaded “ice cream cone anti-pattern”:



Inverted pyramid of testing.

As you can see, it is the *testing* pyramid inverted. In this context the focus is on many manual and *end-to-end* tests. This leads to multiple problems, the main one being that the cost of testing the system skyrockets, this is because without many integration and unit tests (and often with none) it becomes tremendously complex to determine where the problems are when the higher level tests fail.

Unit testing

“We get paid to make software that works, not for testing” - Kent Beck.

Unit testing is not a new concept in the world of software development. Already in the decade of the 70s, when the Smalltalk language emerged, it was talked about, although that was different to how we do it today.

We owe the popularity of unit testing today to Kent Beck. He first introduced it to the Smalltalk language and then made it mainstream in many other programming languages. Thanks to him, unit testing has become an extremely useful and indispensable practice in software development. But what exactly is a unit test?

According to wikipedia: “unit testing is a software testing method by which individual units of source code are tested to determine whether they are fit for use”. The unit of source code being a function or a class.

From my point of view, that definition is incomplete. The first thing that I find strange is that the word “automated” does not appear in the definition. But also, a class is an organizational structure that usually includes several units of code. Normally, to test a class we will need several unit tests, unless we have classes with only one method, which, as we already discussed in the single responsibility chapter, is usually not a good idea. Perhaps a more complete definition would be:

A unit test is a small program that checks automatically that a unit of software has the expected behavior. To do this, it prepares the context, executes said unit, and then verifies the result obtained through one or more assertions that compare it with the expected result.

Characteristics of the unit tests

A few years ago, most developers did not do any unit testing. This still happens today, but not as much. If this is your case, do not worry, since in this section we will bring you up to date. At the other extreme we have the dogmatists of unit testing, who are

for 100% code coverage. This is a mistake, as the test coverage metric only indicates which lines of code have been run (and which have not) at least once when passing unit tests. This usually leads to poor quality tests that do not provide much security, and in this context quantity tends to take precedence over quality. For this reason,

code coverage should never be a goal in itself.

As is almost always the case, in balance there is virtue. For this reason, it is essential to take into account the phrase by Kent Beck with which we open the chapter: “We are paid to write code that works, not for testing.” That is why we must focus on writing quality tests, which in addition to validating useful elements, are:

- **Fast:** The tests have to be executed as quickly as possible, so as to be able to execute them as often as is appropriate and not waste time.
- **Isolated:** It is very important that the status of each test is totally independent and does not affect the others.
- **Atomic:** each test must prove one thing and be small enough.
- **Easy to maintain and extend:** As Edsger Dijkstra says, simplicity is a prerequisite for reliability. Keeping the tests simple is essential to maintaining and extending them.
- **Deterministic:** The results must be consistent, that is, they must always produce the same output from the same starting conditions.
- **Legible:** A legible test is one that clearly reveals its purpose or intent. Therefore, the choice of a self-explanatory name is essential.

Anatomy of a unit test

As a general rule, the tests should have a very simple structure based on the following three parts:

Arrange: in this part of the test we prepare the context, in order to be able to carry out the test. For example, if we test a method of a class, we will first have to instantiate that class to test it. In addition, a part of the preparation can be contained in the `SetUp` method (Before in the case of Jest), if it is common to all the tests in the class.

Act: we execute the action that we want to test. For example, invoke a method with some arguments.

Assert: we verify if the result of the action is as expected. For example, the result of the previous method invocation has to return a certain value.

These three parts are identified with the acronym AAA which helps to remember them, **A**rrange, **A**ct, **A**ssert.

Let's see an example: {title="Anatomy of a unit test", lang=javascript}

```
1 test('return zero if receive one', () => {
2   //Arrange
3   const n = 1;
4
5   //Act
6   const result = fibonacci(n);
7
8   //Assert
9   expect(result).toBe(0);
10 });
```

In this example we have defined a test as we would do it with the framework Jest (we will see this in the next chapter). To do this we have used the test function, which receives two parameters. The first is a string with the description of the test and the second a callback.

The callback contains the logic of the test itself. In the first part of the callback we have the “arrange”. In this case, we initialize a constant with the parameter that we are going to pass to the function we want to test. Next, we have the act where we execute the fibonacci function for the value of n, and store it in the variable result. Finally, we have the part of the assertion where we verify, through the “expect” method of Jest and the matcher toBe, whether the value of the result is what we expected, in this case zero. We will delve into the different assertions that Jest provides us in the next chapter.

We are not always going to have the three parts of the test clearly differentiated, in tests as simple as this we can often find everything in a single line:

```
1 test('return zero if receive one', () => {  
2   expect(fibonacci(1)).toBe(0);  
3 });
```

Personally, even if they are simple tests, I recommend that you try to respect the triple A structure, as it will help you to keep your tests easily readable.

Jest, the definitive JavaScript testing framework

A testing framework is a tool that allows us to write tests easily, it also provides us with an execution environment that allows us to get information from them in a simple way.

Historically JavaScript has been one of the languages with the most testing frameworks and libraries, but at the same time it is one of the languages with the least testing culture among the members of its community. These automated testing

frameworks and libraries include Mocha, Jasmine and Jest, among others. We are going to focus on Jest, since it simplifies the process by integrating all the elements we need to carry out our automated tests.

Jest is a testing framework developed by the Facebook team based on RSpec. Although it was created in the context of React, it is a general testing framework that we can use in any situation. It is a flexible, fast framework with a simple and understandable output, which allows us to complete a fast feedback cycle with maximum information at all times.

Features

Its main features include:

- Easy installation.
- Immediate feedback with 'watch' mode.
- Easy to configure testing platform.
- Fast and sandboxed execution.
- Integrated code coverage tool.
- Introduces the concept of Snapshot testing.
- Powerful mocking library. Works with TypeScript in addition to ES6

Installation and configuration

As in the other examples in the book, we will continue to use [Repl.it](https://repl.it)²² to show the interactive examples. However, I think it is important that we are clear about how to install and configure Jest, in case we want to launch the examples in our local environment.

As with any other JavaScript project, we can install Jest using Yarn or NPM. For our examples we will use NPM. The first thing we need to do is create a directory for the project and then run *npm init* to initialize the project:

```
1 mkdir testing_1
2 npm init
```

Once the project is initialized, we are going to install the dependencies. In this case, in addition to the Jest dependencies, we are going to install *ts-jest* and TypeScript as it simplifies the use of Jest with ES6 and higher. In this way, the project also works for those who prefer TypeScript.

```
1 npm install --save-dev jest typescript ts-jest @types/jest
```

After installing the dependencies we must execute the *ts-jest* configuration command:

```
1 npx ts-jest config:init
```

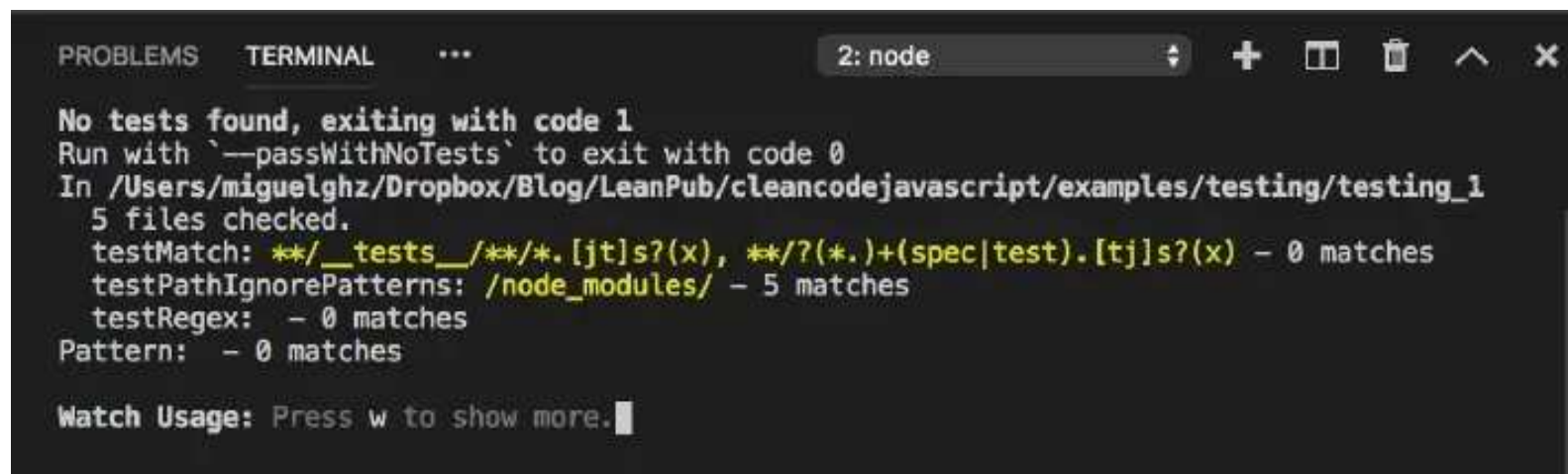
With all the tooling prepared, if we want to run the tests from npm, we just have to add the following to the scripts section of the package.json file:

²²[http://repl.it/](https://repl.it)

```
1 "scripts": {  
2   "test": "jest",  
3   "test:watch": "jest --watchAll"  
4 }
```

The first script (`npm test`) will run the test suite in default mode, that is, it will run the tests and Jest will close. The second, `npm run test:watch`, will execute the tests but will remain in “watcher” mode and every time we make a change in the Jest code it will automatically run the tests again.

Of course, if we try to run some of these scripts, the execution will fail, as we have not yet created any tests in the project.



```
PROBLEMS  TERMINAL  ...  2: node  +  [icon]  [icon]  ^  X  
No tests found, exiting with code 1  
Run with `--passWithNoTests` to exit with code 0  
In /Users/miguelghz/Dropbox/Blog/LeanPub/cleancodejavascript/examples/testing/testing_1  
5 files checked.  
testMatch: **/_tests_/**/*.[jt]s?(x), **/?(*.)+(spec|test).[tj]s?(x) - 0 matches  
testPathIgnorePatterns: /node_modules/ - 5 matches  
testRegex: - 0 matches  
Pattern: - 0 matches  
Watch Usage: Press w to show more.
```

Jest failed execution

You can download the ready-to-use configuration from [our repository](https://github.com/softwarecrafters-io/typescript-jest-minimal)²³ on GitHub.

Our first test

Before creating our first tests, we are going to create the following folder structure:

```
1 src  
2 | --core  
3 | --tests
```

Inside the core folder, we will create the code with our “business rules” and the associated tests in the tests directory.

²³<https://github.com/softwarecrafters-io/typescript-jest-minimal>

Next, we are going to create a simple example that allows us to verify that we have made the configuration correctly. To do this, we create the *sum.ts* file inside the *core* directory with the following code:

```
1 export const sum = (a, b) =>
2   a + b;
```

Once the code to be tested has been created, we are going to create the associated test. To do this, we add the file *sum.test.ts* inside the test directory with the following:

```
1 import {sum} from '../core/sum'
2
3 test('should sum two numbers', () => {
4   //Arrange
5   const a = 1;
6   const b = 2;
7   const expected = 3;
8
9   //Act
10  const result = sum(a,b)
11
12  //Assert
13  expect(result).toBe(expected);
14 });
```

In the test, we simply execute our *sum* function with some input values and wait for it to return an output result. Simple, right? Let's run the tests again to familiarize ourselves with the output:

```
PASS src/tests/sum.test.ts
  ✓ should sum two numbers (2ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.472s, estimated 1s
Ran all test suites.

Watch Usage: Press w to show more.
```

Jest failed execution

Great! We already have our first test working correctly!

Assertions

Assertions are functions provided by the testing frameworks, Jest in our case, to verify if the value expected by the automatic test really matches the value obtained. Obviously, if the value matches, the test will pass, otherwise it will fail.

In the test we saw in the example we used the *toBe* assertion which verifies that two values are equal, and to do this, it is based on `Object.is()`. It is important that you keep this in mind because, in the case of comparing objects or collections, you will get unexpected results.

```
1 expect(1 + 2).toBe(3)
```

If we want to check that they are different, we can use the matcher `toBe` preceded by `not`. This approach is interesting as it adds semantics to our tests.

```
1 expect(1 + 2).not.toBe(4)
```

To verify objects, you must use the `toEqual` assertion, as it performs a deep comparison that correctly checks each field in the structure.

```
1 test('equality of objects', () => {  
2   const data = { one: 1 };  
3   data['two'] = 2;  
4   const expected = { one: 1, two: 2 }  
  
5  
6   expect(data).toEqual(expected);  
7 });
```

In addition to the asserts of equality that we have seen, Jest offers us other interesting ones such as:

- **toBeNull**: validates that the value is null.
- **toBeGreaterThan**: Validates that a numeric value is greater than a specified number. In addition to this we have many other comparators for numerical values that can provide semantics to our tests.
- **toMatch**: validates a string through a regular expression.
- **toContain**: checks that an array contains a specific value.
- **toThrow** - Checks that an exception was thrown.

The reference to the complete list of assertions can be found at this [link](https://jestjs.io/docs/en/expect)²⁴.

Organisation and structure

Structuring our tests well is a basic aspect that determines how easy they are to read and maintain. As a general rule, tests are usually grouped based on a common context, such as, for example, a specific use case. These groups are known as suites.

Jest allows us to define these suites both at the file level and at the context level. At the file level, we must bear in mind that the name must respect one of these two formats: `.spec`. Or `.test..` This is necessary for Jest to detect them without having to modify the configuration. Also, in order to maintain good readability, ideally the preceding suffix should be the name of the file that contains the code we are testing.

On the other hand, within the files themselves we can group the tests in contexts or *describes*. In addition to allowing us to have multiple contexts, Jest allows us to nest

²⁴<https://jestjs.io/docs/en/expect>

them. Although as a general rule, we should avoid nesting in more than two levels, as it hinders the readability of the tests.

Definition of contexts

```
1 describe('Use Case', () => {  
2   test('Should able to do something...', () => {});  
3 });
```

State management: before and after

On many occasions, when we have several tests for the same component, we find ourselves in a context in which we must initialize and/or end the state of said

component. To facilitate this, RSpec-based frameworks provide us with `setup` and `tearDown` methods, which are executed before and after, respectively. In turn, they can be executed before or after each of the tests, or at the beginning and at the end of the entire suite. In the case of Jest these methods are:

```
1 describe('Use Case', () => {  
2   beforeEach(() => {  
3     //run before each test  
4   });  
5  
6   afterEach(() => {  
7     //run after each test  
8   });  
9  
10  beforeAll(() => {  
11    //run before all test  
12  });  
13  
14  afterAll(() => {  
15    //run after all test  
16  });  
17
```

```
18     test('Should able to do something...', () => {  
19  
20     });  
21 });
```

Code coverage

As we mentioned in the previous chapter, code coverage is a metric that indicates the percentage of our code that has been executed by unit tests. Although it can be a dangerous metric, especially if it is used as an indicator of quality (since it could lead to the opposite), it can be useful at a guidance level.

Obtaining this information with Jest is very simple, since we only have to add the `-coverage` flag. If we update the scripts section of our package.json it will look like this:

```
1 "scripts": {  
2   "test": "jest",  
3   "test:watch": "jest --watchAll",  
4   "test:coverage": "jest --coverage"  
5 }
```

If we execute it using the command `npm run test: coverage`, Jest will show us a table as a summary that looks similar to the following:

```

PASS src/tests/sum.test.ts
  ✓ should sum two numbers (4ms)

File      % Stmts   % Branch   % Funcs   % Lines   Uncovered Line #s
-----
All files  100       100        100       100
sum.ts    100       100        100       100

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.297s

```

Code coverage

TDD - Test Driven Development

Test Driven Development (TDD) is a software engineering technique for actually designing software. As its name suggests, this technique directs the development of a product through writing tests, generally unit ones.

The TDD was developed by Kent Beck in the late 1990s and is part of the extreme programming methodology. Its author and the followers of TDD claim that this technique achieves a code that is more tolerant to change, more robust, more secure, cheaper to maintain and, once you get used to applying it, it even promises a greater speed when developing.

The three laws of TDD

Robert C. Martin describes the essence of TDD as a process that meets the following three rules:

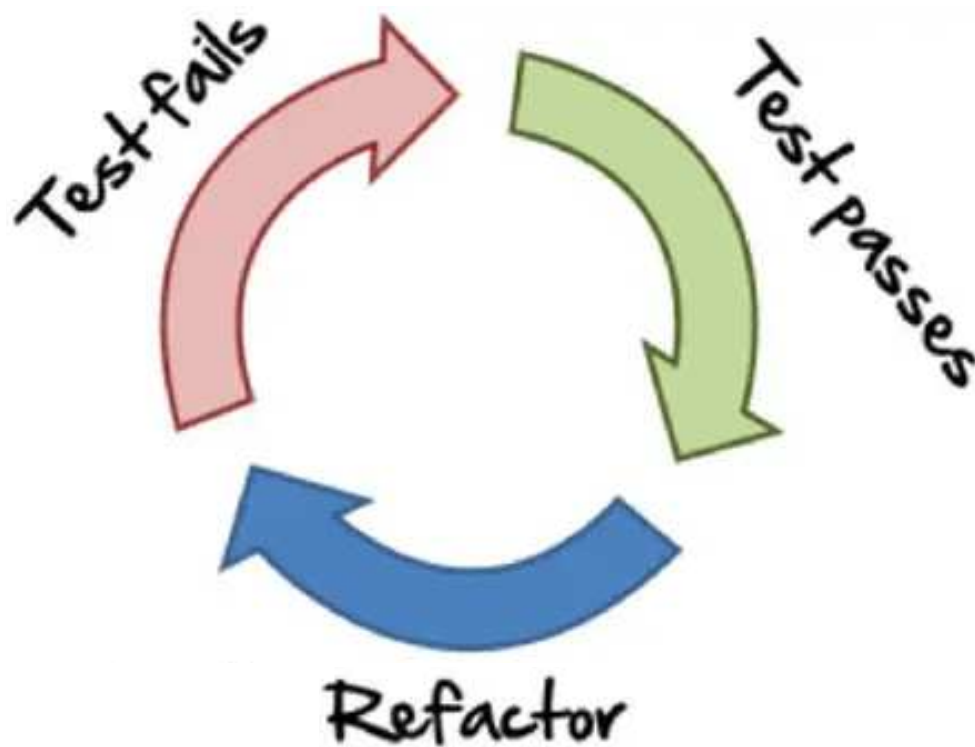
- You will not write production code without first writing a test that fails.
- You will not write more than one unit test that fails. (and not compiling is failing).
- You will not write more code than necessary to pass the test.

These three laws result in the repetition of what is known as the *Red-Green-Refactor* cycle. Let's see what it consists of:

Red-Green-Refactor cycle

The *Red-Green-Refactor* cycle, also known as the TDD algorithm, is based on:

- **Red:** Write a test that fails, that is, we have to create the test before writing the implementation. Unit tests are usually used for this, although in some contexts it may make more sense to do TDD with integration tests.
- **Green:** Once the test that fails is created, we implement the minimum code necessary for the test to pass.
- **Refactor:** Finally, after getting our code to pass the test, we examine it to see if there are any improvements we can make.
- Once we have closed the cycle, we start over with the next requirement.



Cycle Red-Green-Refactor

There are two main benefits to this way of programming. The first and most obvious is that we get a code with good test coverage, which is positive up to a certain point. Remember, we get paid to write code that works, not to test everything.

The second benefit is that writing the tests first helps us to design the API that our artifact will have, since it forces us to think about how we want to use it. This will often generate artifacts with well-defined responsibilities and low coupling.

TDD as a design tool

When Kent Beck developed this methodology, he did so by focusing on the second of the benefits described in the previous section, that is, in TDD as a software design tool that helps us to obtain better code, not to get more tests. To do this, once we have a list of the first requirements to be satisfied by the product, these steps should be followed:

1. Choose a requirement.
2. Write a test that fails.
3. Create the minimum implementation for the test to pass.
4. Run all the tests.
5. Refactor.
6. Update the list of requirements.

In this last step, when we update the list of requirements, in addition to marking the implemented requirement as completed, we must add any new requirements that may have appeared.

Normally, when we develop a software product, the requirements are not completely defined from the beginning, or they change in the short and medium term, either because they are discarded, modified or because new ones arise. TDD fits very well with these types of scenarios since, in addition to adding tests that assess whether our design meets the specified requirements, it also helps to discover new cases that had not been previously detected. The latter is known as **emergent design**.

This is why many of its followers think the last “D” in TDD should stand for design instead of development.

Implementation strategies, from red to green

Perhaps one of the most delicate parts of applying TDD as a design tool is at the point when we already have a test that fails and we must create the minimum implementation for the test to pass. To do this, Kent Beck, outlines a set of strategies,

also known as green bar patterns, that will allow us to advance in small steps toward solving the problem.

Fake implementation

Once the test fails, the fastest way to get the first implementation is by creating a fake that returns a constant. This will help us to progress little by little in solving the problem, because when the test has been passed we will be ready to face the next case.

The best way to understand this concept is with a practical exercise. The exercise is simple, we are going to build a function that receives an integer number n as a parameter and returns the n -th Fibonacci number. Remember the Fibonacci sequence begins with 0 and 1, and the following terms are always the sum of the previous two:

n	0	1	2	3	4	5	6	7	8	9	10	11	12
F_n	0	1	1	2	3	5	8	13	21	34	55	89	144

Fibonacci sequence.

Looking at the table above, we can see that the *edge* cases are 0 and 1, in addition to being the easiest to implement. Let's start by creating the test for $n = 0$:

```
1 describe('Fibonacci should', () => {
2   it('return zero if receive zero', () => {
3     expect(fibonacci(0)).toBe(0);
4   });
5 });
```

The most obvious fake implementation that allows the test to pass is to make the fibonacci function return 0 as a constant:

```
1 function fibonacci(n) {
2   return 0;
3 }
```

You can access the interactive example from [here](https://repl.it/@SoftwareCrafter/TDD-Fibonacci)²⁵.

²⁵<https://repl.it/@SoftwareCrafter/TDD-Fibonacci>

Once we have the first test running, the idea is to gradually transform the constant into an expression. Let's see it in the example, to do this we should first create a test for the following obvious case, $n = 1$;

```
1 it('return one if receive one', () => {  
2   expect(fibonacci(1)).toBe(1);  
3 });
```

We already have the next test failing. The next obvious step is to write a small expression with a conditional for an input with $n = 0$ returns 0 and for $n = 1$ returns 1:

```
1 function fibonacci(n) {  
2   if(n ==0)  
3     return 0;  
4   else  
5     return 1;  
6 }
```

You can access the interactive example from [here](#)²⁶.

As you can see, the false implementation technique helps us to progress little by little. You have two main inherent advantages, the first is on a psychological level, since it makes it easier to have some tests in green, instead of in red, and that allows us to take small steps towards the solution. The second advantage has to do with scope control, since this practice allows us to keep the focus on the real problem, avoiding lapsing into premature optimizations.

Triangulate

Triangulate, or the triangulation technique, is the natural step that follows the fake implementation technique. Furthermore, in most contexts, this fake implementation is actually part of triangulation, as follows:

1. Choose the simplest case that the algorithm must solve.

²⁶<https://repl.it/@SoftwareCrafter/TDD-Fibonacci-1>

2. Apply *Red-Green-Refactor*.
3. Repeat the previous steps covering the different cases.

To understand how triangulation works, we will continue to develop the Fibonacci example, which, in part, we have already begun to triangulate. The next case is for $n = 2$.

```
1 it('return one if receive two', () => {  
2   expect(fibonacci(2)).toBe(1);  
3 });
```

You can access the interactive example from [here](#)²⁷.

This time the test passes, therefore, our algorithm also works for $n = 2$. The next step is to check what happens for $n = 3$.

```
1 it('returns two if receive three', () => {  
2   expect(fibonacci(3)).toBe(2);  
3 });
```

As we expected, the test fails. This step will help us get closer to implementing a more generic solution, because we can create a false implementation for $n = 3$ and add another conditional that returns 1 for $n = 1$ and $n = 2$.

```
1 function fibonacci(n) {  
2   if(n == 0)  
3     return 0;  
4   if(n == 1 || n == 2)  
5     return 1;  
6   else  
7     return 2;  
8 }
```

You can see the interactive example from [here](#)²⁸

Now that we have got all the tests to pass, let's check what happens for $n = 4$:

²⁷<https://repl.it/@SoftwareCrafter/TDD-Fibonacci-2>
²⁸<https://repl.it/@SoftwareCrafter/TDD-Fibonacci-3>

```
1 it('returns three if receive four', () => {
2   expect(fibonacci(4)).toBe(3);
3 });
```

By now, you may have realized that it would be easier to write the obvious implementation rather than to keep making decision branches:

```
1 function fibonacci(n) {
2   if(n == 0)
3     return 0;
4
5   if(n == 1 || n == 2)
6     return 1;
7
8   else
9     return fibonacci(n - 1) + fibonacci(n - 2);
10 }
```

In this step, our algorithm works for any value of n, although we can still refactor it to remove duplicates and give it a more functional look:

```
1 function fibonacci(n) {
2   const partialFibonacci = (n) =>
3     n == 1
4       ? 1
5       : fibonacci(n - 1) + fibonacci(n - 2)
6
7   return n == 0
8     ? 0
9     : partialFibonacci(n)
10 }
```

You can access the interactive example from [here](https://repl.it/@SoftwareCrafter/TDD-Fibonacci-4)²⁹.

²⁹<https://repl.it/@SoftwareCrafter/TDD-Fibonacci-4>

With this last step we have solved the Fibonacci algorithm applying a functional approach and using triangulation. Perhaps a hypothetical next step would be to eliminate the tests for $n = 3$, $n = 4$ and $n = 5$, since at this point they do not provide much value, and then create a test that verifies the algorithm generating a random

number greater than 2 each time it runs.

As you can see, triangulation is a very conservative way of applying TDD, it makes sense to use it when we are not clear about the obvious implementation of the solution.

Obvious implementation

When the solution seems very simple, it is best to write the obvious implementation in the first iterations of the Red-Green-Refactor loop.

The problem with this appears when we rush, believing that it is a simple problem, when in fact it is not, because it has, for example, some edge case which we hadn't thought about.

TDD Limitations

No matter how many inherent benefits it has (or are promised), the TDD technique should not be understood as a religion or a one-size-fits-all magic formula. Following TDD to the letter and in all contexts does not guarantee that your code will be any more robust, secure or tolerant to change. It does not even ensure that you will be

more productive when designing software.

From my point of view, applying TDD isn't appropriate in all contexts. For example, if there is an obvious implementation for a use case, I directly write it and then do the tests. In the case of working on the frontend, I don't even consider doing TDD to design UI components. It is even debatable whether unit tests should be done to test elements of the UI. Developers such as Ward Cunningham have repeatedly commented that it is not convenient to do automated tests on UI, since it is very changeable and the tests frequently become outdated.

My advice is that you try it. Try to apply it in your day to day for a time and then

decide for yourself. In the following chapters we are going to see some katas so that you can keep practicing.

Practical TDD: The FizzBuzz Kata

“Many people are competent in the classroom, but put what they learn into practice and they fail miserably.” - Epictetus

Going from theory to practice is essential. In the development world, if you study a concept and don't put it into practice for a few days with enough spaced repetition, you will probably never internalize it. And so, it is highly recommended that you try to practice as much as you can, and performing katas of code will help you with this goal.

The code katas

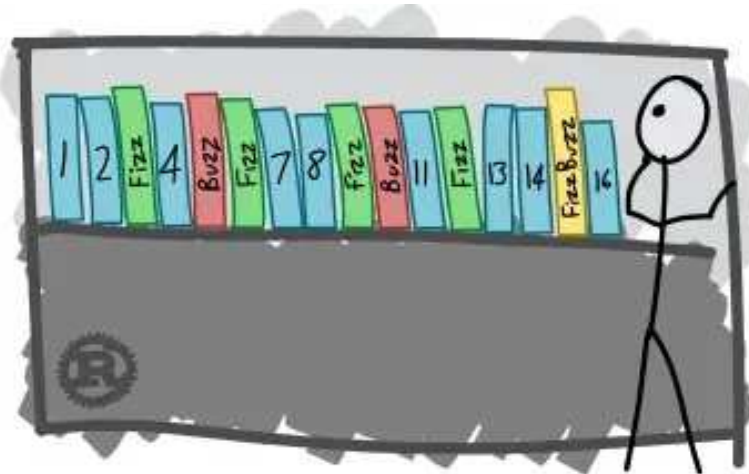
In martial arts, a kata is a set of established movements. They are usually performed solo in order to perfect the foundations of the knowledge of a specific martial art. Although these have an aesthetic component, their purpose is not to represent them on stage, but to train the mind and body on how to react in a given combat situation.

The concept of code kata is a programming exercise in which a problem (with or without restrictions) is exposed, which the developer should try to solve, ideally, using different techniques. The term was first coined by Dave Thomas, co-author of the mythical book *The Pragmatic Programmer*, of which, by the way, they have recently published a special twentieth anniversary edition.

Dave Thomas said that musicians, when trying to improve, are not always playing with the band, but are often doing solo exercises that allow them to refine their technique. The developers spend all day at work programming in real projects, normally solving the same type of problems. Using this simile of musicians, we could say that we are playing in the band all day. After a certain time, this limits our ability to learn, so performing solo katas or going to coding dojos will help us acquire new skills.

The Fizzbuzz kata

The FizzBuzz kata, in addition to being a great exercise to practice TDD, is one of the typical tests you could face in a job interview for a developer position. This exercise has its origin in a children's game for practicing division. This kata began to become popular with developers after Emily Bache and Michael Feathers performed it in the *Programming with the Stars* competition at the 2008 "Agile Conference".



FizzBuzz Kata

Exercise statement

The wording of the kata is as follows: Write a program that displays the numbers from 1 to 100 on the screen, replacing the multiples of 3 with the word Fizz and, in turn, the multiples of 5 with Buzz. For numbers that are multiples of 3 and 5, use the FizzBuzz combo.

Example output:

1, 2, Fizz, 4, Buzz, Fizz, 7, 8, Fizz, Buzz, 11, Fizz, 13, 14, FizzBuzz, 16, 17, Fizz, 19, Buzz
..

Designing the first test

Before writing the first test, I like to check that all the parts are correctly configured. To do this I write a simple test with which I simply check that the testing framework works correctly.

```
1 describe('FizzBuzz', () => {  
2   it('', () => {  
3     expect(true).toBe(true);  
4   });  
5 });
```

You can access the example [from here](https://repl.it/@SoftwareCrafter/TDD-Fizzbuzz-Green)³⁰

Once we are sure that our suite works as expected, we can go about choosing the first test. In TDD this can sometimes be quite difficult, so it is interesting to make a small list, as if it were a black box, with the different cases that our tests should cover:

- For the number one, the result should be one
- For number three, the result should be “fizz”
- For the number five, the result should be “buzz”
- For the number fifteen, the result should be “fizzbuzz”
- For any number divisible by three, the result should be “fizz”
- For any number divisible by five, the result should be “buzz”
- For any number divisible by fifteen, the result should be “fizzbuzz”
- For the rest of the numbers, the result should be the received number itself.

If requirements emerge during the TDD process, it is important to update our list.

Once we have the list of the different cases to cover, we will be in a position to address the problem. To do this we will start with a very simple initial design, in which we will have a function that receives an integer and returns zero. The *fizzBuzz.js* file will contain this function, looking like this:

³⁰<https://repl.it/@SoftwareCrafter/TDD-Fizzbuzz-Green>

```
1 function fizzBuzz(n) {  
2   return 0;  
3 }  
4  
5 module.exports = fizzBuzz;
```

Then we will be able to write the first test. So we start with the first case on our list, in which for number one the result should be one:

```
1 describe('FizzBuzz', () => {  
2   it('should return one if receive one', () => {  
3     const expected = 1;  
4     const result = fizzBuzz(1)  
5  
6     expect(result).toBe(expected);  
7   });  
8 });
```

We run and... red!

We run the test suite and the test fails as expected. This first test in red is important, as it can help us to detect possible errors in the construction of the test. It is usually quite common for the test to fail, not because the implementation code is incorrect, but because we have made a mistake when implementing it.

```
• FizzBuzz > should return one if receive one

expect(received).toBe(expected) // Object.is equality

Expected: 1
Received: 0

   6 |     const result = fizzBuzz(1)
   7 |
>  8 |     expect(result).toBe(expected);
      |                       ^
   9 |   });
  10 | });

at Object.toBe (fizzBuzz-test.js:8:20)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 total
```

We run and ... red!

We go green

Next, we are going to apply the concept of false implementation that we saw in the previous chapter to get the test to pass as soon as possible:

```
1 function fizzBuzz(n) {
2   return 1;
3 }
4
5 module.exports = fizzBuzz;
```

You can access the interactive example [from here](https://repl.it/@SoftwareCrafter/TDD-Fizzbuzz-1-Green)³¹

We run the test and it passes correctly. In these first tests we will have very concrete implementations. As we advance through the solution, we gradually generalize the solution.

³¹<https://repl.it/@SoftwareCrafter/TDD-Fizzbuzz-1-Green>

Adding new tests

Going back to our list, the next case to check is: “For number three the result must be fizz.” Let’s go:

```
1 describe('FizzBuzz', () => {
2   it('should return one if receive one', () => {
3     const expected = 1;
4     const result = fizzBuzz(1);
5
6     expect(result).toBe(expected);
7   });
8
9   it('should return fizz if receive three', () => {
10    const expected = "fizz";
11    const result = fizzBuzz(3)
12
13    expect(result).toBe(expected);
14  });
15 });
```

You can access the interactive example [from here](https://repl.it/@SoftwareCrafter/TDD-Fizzbuzz-2-Red)³²

We run the test suite and the new test fails. It is important to run all the tests to verify the integrity of the implementation that we are carrying out.

³²<https://repl.it/@SoftwareCrafter/TDD-Fizzbuzz-2-Red>

```
FizzBuzz
✓ should return one if receive one (5ms)
✗ should return fizz if receive three (5ms)

• FizzBuzz > should return fizz if receive three

expect(received).toBe(expected) // Object.is equality

Expected: "fizz"
Received: 1
```

Second test in red

Next we write the minimum implementation necessary for the new test to pass without breaking the previous one. For this we can make use of a conditional that returns the value fizz in the case that n is equal to three:

```
1 function fizzBuzz(n) {
2   if(n==3)
3     return "fizz";
4
5   return 1;
6 }
```

Once we have the second test in green, we are going to do the third case: “For number five the result must be buzz”. We write the test:

```
1 it('should return buzz if receive five', () => {
2   const expected = "buzz";
3   const result = fizzBuzz(5)
4
5   expect(result).toBe(expected);
6 });
```

Red. Now we should create the implementation for this case, as in the previous case we can make use of a conditional, this time that returns “fizz” when n is equal to 5:

```
1 function fizzBuzz(n) {  
2   if(n == 3)  
3     return "fizz";  
4  
5   if(n == 5)  
6     return "buzz";  
7  
8   return 1;  
9 }
```

You can access this step [from here](#)³³.

At this point, before tackling the next case and as our knowledge of the problem has matured, it might be interesting to refactor in order to begin to generalize the solution a bit. To do this, in the conditionals, instead of checking if it is a three or a five, we might check if the number is divisible by three or five and return fizz or buzz, respectively. Also, instead of returning 1 for any value of n, we could return the value of n itself:

As you can see in the [interactive example](#)³⁴ the tests are still green.

The next thing to check is the use case in which for n is equal to fifteen the function returns 'fizzbuzz':

```
1 it('should return fizzbuzz if receive fifteen', () => {  
2   const expected = "fizzbuzz";  
3   const result = fizzBuzz(15)  
4  
5   expect(result).toBe(expected);  
6 });
```

If we run the [interactive example](#)³⁵ we can see that this last test is in red while the rest remain green. We are going to correct it: and to do this, as we did in the previous refactor, we are going to check through a conditional that evaluates whether the number received per parameter is divisible by fifteen:

³³<https://repl.it/@SoftwareCrafter/TDD-Fizzbuzz-3-Green>

³⁴<https://repl.it/@SoftwareCrafter/TDD-Fizzbuzz-3-Refactor>
³⁵<https://repl.it/@SoftwareCrafter/TDD-Fizzbuzz-4-Red>

```
1 function fizzBuzz(n) {
2   if(n % 15 == 0)
3     return "fizzbuzz";
4
5   if(n % 3 == 0)
6     return "fizz";
7
8   if(n % 5 == 0)
9     return "buzz";
10
11  return n;
12 }
```

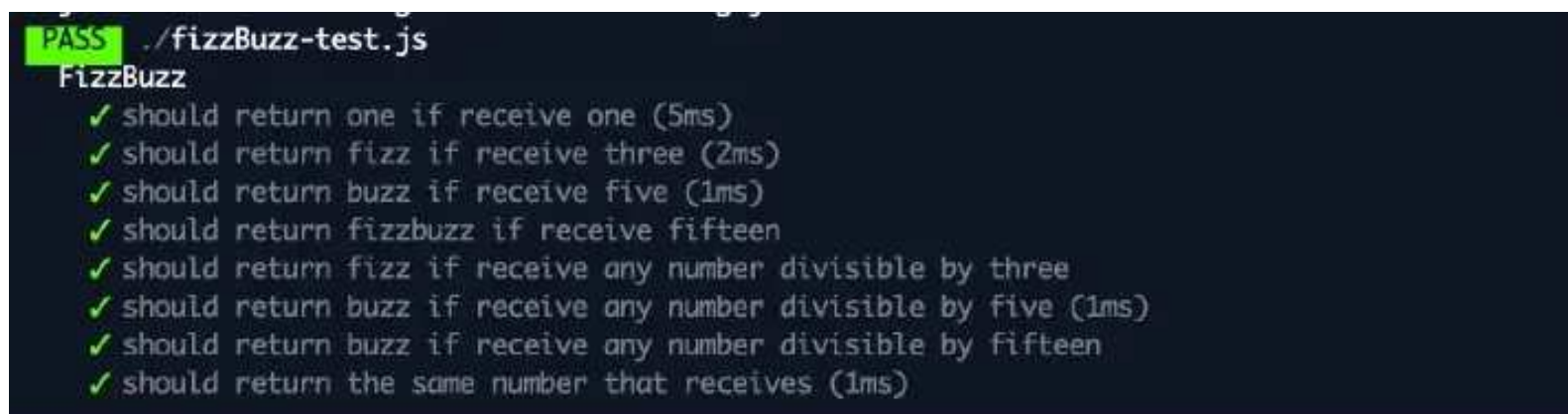
We run the test suite again and the tests pass successfully³⁶. You have probably already realized that we have arrived at a valid solution, which solves the exercise, but we have not verified it yet. To check this, in the next step we are going to write the rest of the missing tests:

```
1 it('should return fizz if receive any number divisible by three', () => \
2   {
3     const expected = "fizz";
4     const result = fizzBuzz(9)
5
6     expect(result).toBe(expected);
7   });
8
9 it('should return buzz if receive any number divisible by five', () => {
10   const expected = "buzz";
11   const result = fizzBuzz(25)
12
13   expect(result).toBe(expected);
14 });
15
16 it('should return buzz if receive any number divisible by fifteen', () \
```

³⁶<https://repl.it/@SoftwareCrafter/TDD-Fizzbuzz-4-Green>

```
17 => {
18   const expected = "fizzbuzz";
19   const result = fizzBuzz(30)
20
21 }
22 expect(result).toBe(expected);
23
24 it('should return the same number that receives', () => {
25   const expected = 4;
26   const result = fizzBuzz(4)
27
28   expect(result).toBe(expected);
29 });
```

If we run the rest of the cases in the [interactive console](#)³⁷ we can verify that our implementation actually complies with all the cases that we have listed:



```
PASS ./fizzBuzz-test.js
FizzBuzz
✓ should return one if receive one (5ms)
✓ should return fizz if receive three (2ms)
✓ should return buzz if receive five (1ms)
✓ should return fizzbuzz if receive fifteen
✓ should return fizz if receive any number divisible by three
✓ should return buzz if receive any number divisible by five (1ms)
✓ should return buzz if receive any number divisible by fifteen
✓ should return the same number that receives (1ms)
```

All tests green

Refactoring the solution, applying pattern matching.

Once we have covered all the possible scenarios, it is the ideal time to refactor, to clean up and add readability to the code we have obtained. In this case, we could add some semantics to it, extracting a lambda function that checks if *n* is divisible by a given number:

³⁷<https://repl.it/@SoftwareCrafter/TDD-Fizzbuzz-5-Green>

FizzBuzz, refactoring.

```
1 function fizzBuzz(n) {
2   const divisibleBy = (divider, n) => n % divider == 0;
3
4   if(divisibleBy(15, n))
5     return "fizzbuzz";
6
7   if(divisibleBy(3, n))
8     return "fizz";
9
10  if(divisibleBy(5, n))
11    return "buzz";
12
13  return n;
14 }
15
16 module.exports = fizzBuzz;
```

You can access the example [from here](#)³⁸.

As we discussed in the functions chapter of the *Clean Code* section, I like to prioritize the declarative style over the imperative, since it allows us to obtain much more expressive functions. Applying pattern matching suits our problem very well. Pattern matching is a structure typically used by statically typed and functional programming languages such as Haskell, Scala or F #.

This structure allows us to check a value against a series of cases. When a case is true, the associated expression is executed and terminated. Ideally, cases allow you to specify not only constant values, but also types, types with specific properties, or complex conditions. Conceptually, it looks like an upgraded switch-case structure.

Although JavaScript does not support pattern matching in its syntax yet ([there is already a proposal for it](#)³⁹), we can use libraries to make up for this lack. In our case we are going to use the npm package called [x-match-expression](#)⁴⁰.

³⁸<https://repl.it/@SoftwareCrafter/TDD-Fizzbuzz-5-Final>

³⁹<https://github.com/tc39/proposal-pattern-matching>
⁴⁰<https://www.npmjs.com/package/x-match-expression>

```
1 import { match } from "x-match-expression";
2
3 export function fizzBuzz(n) {
4   const divisibleBy = divider => n => n % divider === 0;
5
6   return match(n)
7     .case(divisibleBy(15), "fizzbuzz")
8     .case(divisibleBy(5), "buzz")
9     .case(divisibleBy(3), "fizz")
10    .default(n);
11 }
```

You can access the full interactive example [from here](#)⁴¹

As we can see, the code now looks more concise and expressive. Although, if you are not used to the functional style, it can be a bit confusing, so let's explain it in detail. First, we import the *match* function, which will allow us to do the pattern matching itself. Next we have curried the *divisibleBy* function created in the previous step in order to pass it on to each of the cases while applying composition. Finally, the value of *n* is evaluated through *match*, and each of the cases will be executed until one matches the condition passed in the expression. If none of the three cases match, it will return *n* by default.

If you want to try some more, I recommend that you visit the [kata-log.rocks](#)⁴² page, where you will find many exercises with which to continue practicing. Remember:

Practice makes perfect!

⁴¹<https://codesandbox.io/s/fizzbuzz-pattern-matching-tdd-p7oby>

⁴²<https://kata-log.rocks/>

References

- Clean Code: A Handbook of Agile Software Craftsmanship by Robert C. Martin⁴³
- Clean Architecture: A Craftsman's Guide to Software Structure and Design by Robert C Martin⁴⁴
- The Clean Coder: A Code of Conduct for Professional Programmers by Robert C. Martin⁴⁵
- Test Driven Development. By Example by Kent Beck⁴⁶
- Extreme Programming Explained by Kent Beck⁴⁷
- Implementation Patterns by Kent Beck⁴⁸
- Refactoring: Improving the Design of Existing Code by Martin Fowler⁴⁹
- Design patterns by Erich Gamma, John Vlissides, Richard Helm y Ralph Johnson⁵⁰
- Effective Unit Testing by Lasse Koskela⁵¹
- The Art of Unit Testing: with examples in C# by Roy Osherove⁵²
- JavaScript Allonge by Reg “raganwald” Braithwaite⁵³
- You Don't Know JS by Kyle Simpson⁵⁴
- Diseño Ágil con TDD - Carlos Blé⁵⁵
- Repo - Ryan McDermott⁵⁶

- ~~Airbnb guidestyle~~⁵⁷

⁴³<https://amzn.to/2TUywwB>

⁴⁴<https://amzn.to/2ph2wrZ>

⁴⁵<https://amzn.to/2q5xgws>

⁴⁶<https://amzn.to/2J1zWSH>

⁴⁷<https://amzn.to/2VHQkNg>

⁴⁸<https://amzn.to/2Hnh7cC>

⁴⁹<https://amzn.to/2MGmeFy>

⁵⁰<https://amzn.to/2EW7MXv>

⁵¹<https://amzn.to/2VCcsbP>

⁵²<https://amzn.to/31ahpK6>

⁵³<https://leanpub.com/javascript-allonge>

⁵⁴<https://amzn.to/2OJ24xu>

⁵⁵<https://www.carlosble.com/libro-tdd/?lang=es>

⁵⁶<https://github.com/ryanmcdermott/clean-code-javascript>

⁵⁷<https://github.com/airbnb/javascript>

- [From Stupid to SOLID code - Willian Durand⁵⁸](#)
- Conversations with colleagues [Carlos Blé⁵⁹](#), Dani García, [Patrick Hertling⁶⁰](#) y [Juan M. Gómez⁶¹](#).

⁵⁸<https://williamdurand.fr/2013/07/30/from-stupid-to-solid-code/>

⁵⁹<https://twitter.com/carlosble>

⁶⁰<https://twitter.com/PatrickHertling>

⁶¹https://twitter.com/_jmgomez_